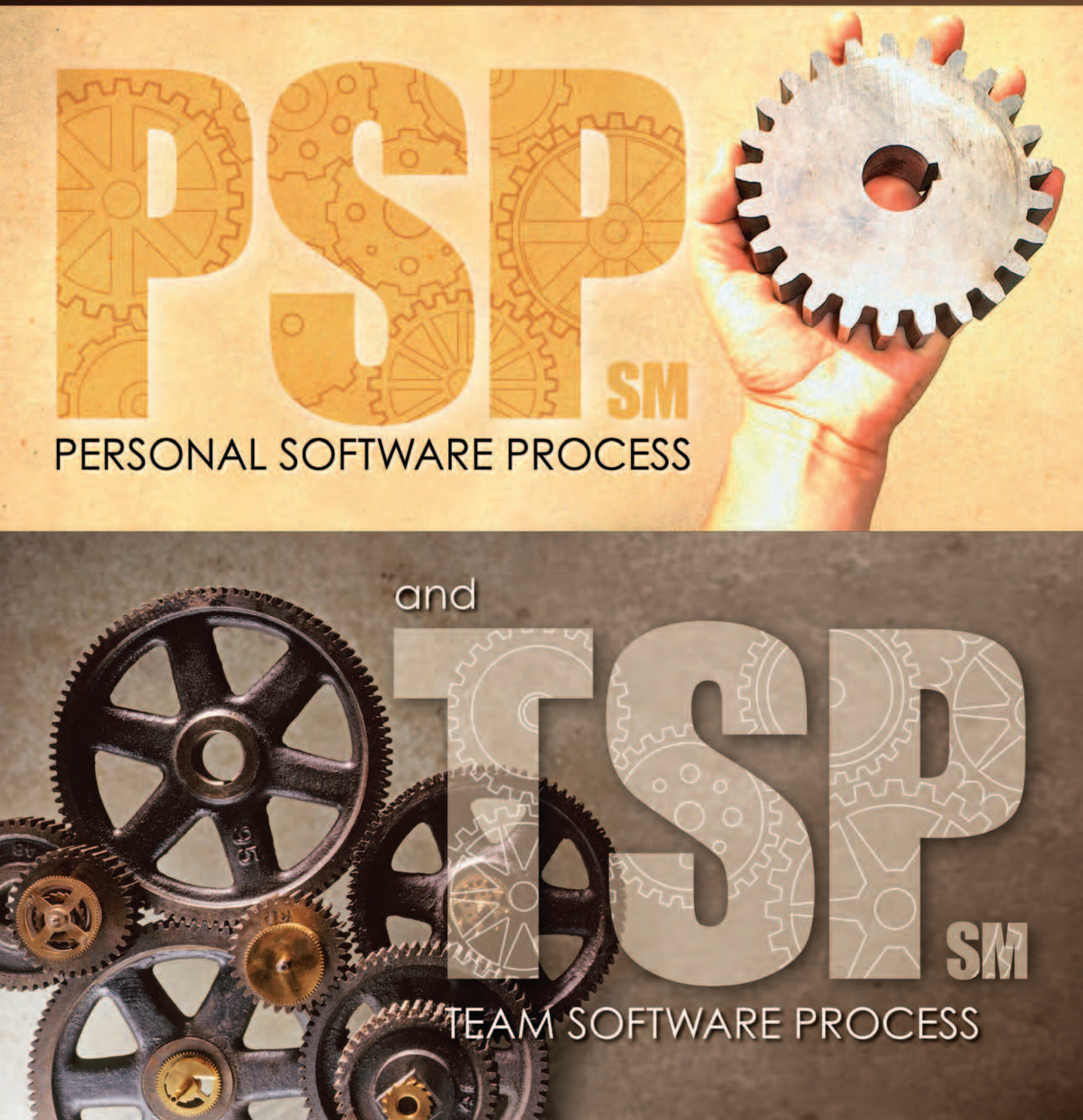
CrossTalk

March 2006 **The Journal of Defense Software Engineering** Vol. 19 No. 3



PSP℠

PERSONAL SOFTWARE PROCESS

and

TSP℠

TEAM SOFTWARE PROCESS

PSP/TSP

Software Engineering Technology

On The Cover
Cover Design by Kent Bingham.

Additional art services provided by Janna Jensen. jensendesigns@aol.com

Departments

CrossTalk

76 SMXG Co-Sponsor	*Kevin Stamey*
309 SMXG Co-Sponsor	*Randy Hill*
402 SMXG Co-Sponsor	*Bob Zwitch*
DHS Co-Sponsor	*Joe Jarzombek*
NAVAIR Co-Sponsor	*Jeff Schwalb*
Publisher	*Brent Baxter*
Associate Publisher	*Elizabeth Starrett*
Managing Editor	*Pamela Palmer*
Associate Editor	*Chelene Fortier-Lozancich*
Article Coordinator	*Nicole Kentta*
Phone	(801) 775-5555
E-mail	crosstalk.staff@hill.af.mil
CrossTalk Online	www.stsc.hill.af.mil/crosstalk

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Air Force (USAF), the U.S. Department of Homeland Security (DHS), and the U.S. Navy (USN). USAF co-sponsors: Oklahoma City-Air Logistics Center (ALC) 76 Software Maintenance Group (SMXG), Ogden-ALC 309 SMXG, and Warner Robins-ALC 402 SMXG. DHS co-sponsor: National Cyber Security Division of the Office of Infrastructure Protection. USN co-sponsor: Naval Air Systems Command (NAVAIR) Software Systems Support Center.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk, providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

S T S C

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail us or use the form on p. 13.

309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CrossTalk does not pay for submissions. Articles published in CrossTalk remain the property of the authors and may be submitted to other publications.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: This Department of Defense (DoD) journal is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, or the STSC. All product names referenced in this issue are trademarks of their companies.

Coming Events: Please submit conferences, seminars, symposiums, etc. that are of interest to our readers at least 90 days before registration. Mail or e-mail announcements to us.

CrossTalk Online Services: See <www.stsc.hill.af.mil/crosstalk>, call (801) 777-0857 or e-mail <stsc.webmaster@hill.af.mil>.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

Software Product Development: Transforming Art to Science

The functional capabilities of today's aircraft and weapon systems are increasingly dependent on the software resident in these systems. Expanding the functionality to meet the demands of an ever-changing environment drives a seemingly insatiable demand for software development resources. The realities of our fiscal and personnel environments preclude a one-for-one approach to meeting those demands. Like many other organizations, the Naval Air Systems Command (NAVAIR) has embarked on a journey to transform the way it acquires and develops software. Nothing less than a transformation will yield the efficiencies we need to support our customers: the sailors and Marines who utilize our products to protect and defend our nation.

The military, civilian, and contractor teams supporting NAVAIR have produced software products that have met user expectations for many years. Their dedication and hard work has allowed successful deployment of very complex weapon systems. However, I would have to characterize the efforts as more art than science. Many of our most significant releases have been based on the heroic actions of team members. While laudable, such an approach is not sustainable in a steady-state environment, much less in one of significantly increasing demands.

Our quest for transformation is based on some important changes. Two of them are structural in nature. The other, tied closely with the structural changes, is cultural. The initial structural change – already under way – groups our existing 52 standalone product teams into four Mission Area Teams (MATs), each with a single leader responsible for working with many customers and sponsors to deliver products efficiently by utilizing shared resources. This specific change is driving the largest cultural change. Prior to this transformation, each separate product team leader carried full resource and product responsibility. This arrangement did little to enhance efficiency across the teams as resource sharing could lead to perceived increases in risk to the separate product teams.

The second structural change is more germane to CROSSTALK. NAVAIR has decided to embrace the Software Engineering Institute's (SEI℠) Capability Maturity Model® Integration (CMMI®) model as overall architecture to guide process improvement within the MATs. NAVAIR has a long history of utilizing SEI models, and has achieved significant success in process improvement using the CMM® for Software (SW-CMM), but it has been a fragmented approach dependent on the process improvement philosophy of each individual team. Within CMMI, processes are developed at the organizational level with tailoring guides to allow individual project teams within the organization to utilize the processes. We have established the MATs as the CMMI organizational focus, and are working to document the organizational level processes and tailoring guides for each applicable process area. This single structural change will allow much more flexibility for resource sharing between teams as each MAT member will utilize the same processes regardless of his or her specific project team. This overarching process improvement culture should bring steadily increasing rigor to our software development practices, allowing us to accurately predict and execute resource requirements and project risks, precluding the need for continuous heroic efforts.

Each NAVAIR product team selected combinations of tools that met its specific needs to do software process improvement, including SW-CMM, CMMI, Earned Value Management System, High Performance Organization training, and Team Software Process℠ (TSP℠). Since the theme of this edition of CROSSTALK is Personal Software Process℠/TSP, I would like to reference some specific examples of how TSP has significantly helped accelerate organizational software process improvement in NAVAIR. SEI data shows an average of six years to achieve SW-CMM Level 4. At NAVAIR, we proudly point to three organizations at multiple locations that successfully used TSP to achieve SW-CMM Level 4 in less than three years: AV-8B (2003), P-3C (2004), and E-2C (2005)!

The articles within this issue are intended to give you a flavor of the successful use of TSP to further software process improvement efforts. I hope you enjoy learning from your colleagues' past efforts and future plans.

Terrence Clark
Director, Software Engineering
Naval Air Systems Command

Using TSP With a Multi-Disciplined Project Management System

Timothy A. Chick
Naval Air Warfare Center

The Team Software Process℠ (TSP℠) provides an extraordinary amount of data, including project planning and tracking data in terms of task hours and earned value, but it does not provide a mechanism for incorporating the plans of multiple teams, which do not all use TSP. In today's world of large and complex systems, the project must consist of multiple disciplines such as software engineers, system engineers, hardware engineers, domain experts, test engineers, and other support personnel. To plan and track a multi-discipline project, a consolidated plan must be created and tracked. Not all disciplines are able or willing to use TSP, so traditional project planning and tracking methods must be used for the overall project.

Standard Team Software Process℠ (TSP℠) for a single project is designed for software teams of between three and 15 software engineers. TSP does not address other disciplines or how to integrate the plans and schedules of the many individuals needed to develop, build, or maintain a large complex system. Because it does not address how to integrate the plans and schedules of a large system, it also does not address how to manage or track such a large project.

This is the dilemma I found myself in a few years ago when my organization decided to begin using TSP for its software work. In addition to software engineering, a typical project for my group includes systems engineering, independent verification and validation, domain expertise, flight testing, and multiple support functions such as configuration management and quality assurance. Of all these disciplines, only the software engineering group was able or willing to use TSP, so traditional project planning and tracking methods were needed for the overall project.

Why Have a Consolidated Plan?

A consolidated plan forces the many disciplines' practitioners to think through their approach and make decisions about how to proceed. It forces the disciplines' practitioners to think outside their proverbial bubbles and consider how the work they perform impacts other groups. Network-based schedules can then be created to identify the interdependencies of activities and the impacts of late or early starts.

Once the schedule is developed, a critical path can be identified and *what if* exercises can be performed. In addition, resources – including facilities, equipment, and personnel – can be identified and tracked. The earned value (EV) method for each task should be determined during the development of the plan. The consolidated plan will provide a vehicle to facilitate executive and customer review. The process of developing a consolidated plan will often identify missing work in the individual team's plans, thereby identifying potential problems early.

A consolidated project plan should define how, when, by whom, and for how much. It should adequately reflect contract milestones, establish meaningful indicators to measure work progress, and allow for the identification of specific activities and events that contribute to schedule variances. Without addressing the interdependencies of the many disciplines, you cannot adequately address these questions.

> *"The process of developing a consolidated plan will often identify missing work in the individual team's plans, thereby identifying potential problems early."*

TSP Versus Traditional Project Planning and Tracking

TSP encapsulates many of the elements of traditional project planning and tracking such as work breakdown structure (WBS) or size summary (SUMS), which is a deliverable-oriented, hierarchical decomposition of the work to be executed [1]. TSP also addresses things like tasks and resource allocation at a much more detailed level than most common project plans. However, TSP does not fully encapsulate other concepts of traditional project planning and tracking such as (1) resource dictionaries, which include labor category, rate (cost/staff hour), and resource availability; (2) tasks with associated logic; (3) Gantt/Pert graphs; and (4) critical path analysis. This article will only scratch the surface of some of these concepts; for more details refer to [1].

After many years of experimenting and using different techniques for project planning and tracking, my organization has developed the following guidelines to consider when developing a consolidated plan:

- WBS – develop to two or three levels:
  - Determine EV method.
    - TSP tasks use percent complete as defined in the section "Converting TSP EV Into EVMS EV" (see page 6).
    - Non-TSP tasks use 50/50 as the preferred method; 0/100 is only used for tasks less than or equal to a one-month duration or reporting period.
    - Work activities are not to exceed two-month durations or two reporting period durations.
  - Tasks should be discrete, resulting in a product or measurable result.
  - Resources should be assigned with budgeted hours or expense.
- Limit level-of-effort (LOE) activities to less than 10 percent of total effort.

Earned Value Management

EV is the budgeted value for an element of work that has been completed, with that value determined from what had initially been planned for accomplishing that element of work. EV techniques have been developed to provide multiple ways to measure accomplishment that best fits

℠ Team Software Process and TSP are service marks of Carnegie Mellon University.

the work being accomplished [2]. TSP uses the EV technique 0/100, which allows no credit of a given task until the task is completed. No value is earned for starting a task or for partially completing a task.

This planning technique should be limited to tasks that are planned to start and complete within the same reporting period. This method works well using TSP because TSP breaks the work task down to such granularity that a task is completed every week, which is the reporting period of a TSP team. A TSP team meets weekly to discuss individual status and how it impacts the team's commitments.

I have found, however, that this method usually does not work very well for a large complex project, which uses traditional project planning and tracking because developing such a detailed plan at the project level is usually not realistic. It is very difficult to break tasks down to extremely small elements and still be able to accurately apply duration and logic while still accounting for any interdependencies with other disciplines.

For example, many projects report EV monthly. In this case, 0/100 would be a good method of EV for tasks that are less than one month in duration. If you used 0/100 for tasks that had duration longer than the reporting period, then you would have spikes in your EV. This makes it very difficult to determine if the project is progressing as planned or if corrective action is required.

Other EV methods not used by TSP are 50/50, percent complete, and LOE. The 50/50 technique is typically used when the task begins in one reporting period and completes in the next. The 50/50 technique credits 50 percent of the EV when a task is started and 50 percent of the EV when the task is completed. By receiving EV for starting and completing the task, this method allows a project to show progress during both reporting periods.

The least desirable of the discrete EV techniques is the percent-complete method. Percent complete allows an estimated percent completed to be assigned for a given reporting period. Unlike 0/100 and 50/50, this method can be extremely subjective. For example, if the estimated percent completed is based on opinion rather than on an objective evaluation of the work completed and the work remaining, then the EV assigned to the task may be grossly inaccurate.

Certain tasks are difficult to quantify in terms of work accomplished; these tasks are referred to as LOE. Because the EV will always equal the budget, there will never be any schedule variance. This method should only be used for tasks in which no tangible product is developed such as project management, clerical support, etc. This method would be used for many of the off-task items identified in TSP such as meetings, phone calls, or anything else not directly related to the immediate activity.

In my experience, groups that are new to EV try to use this method the most because it is the easiest to define and track. The problem is that it does not usually give an accurate picture of how the project is progressing. In fact, when LOE is greater than 10 percent of total effort, the EV becomes skewed to the point that it is very difficult to determine if the project is on schedule or on cost, and if the project will be able to make its commitments.

Developing a Project's EV Management System With TSP as an Input

Now that I have gone over why a consolidated plan is needed and some of the differences between TSP and traditional project planning and tracking, how do you make the leap from a TSP launch to a consolidated project EV management system (EVMS)?

The TSP launch provides most of the core data elements needed as input into a consolidated EVMS. It provides estimates, tasks to be performed, durations, sequential logic and milestones, and assigns resources to tasks. The data hours found in TSP are the *task hours* that resources will actually work to complete the task identified in TSP. The EVMS system captures *staff hours*, which are all of the hours the resources work in each period. Therefore, a conversion of the data is necessary. The EVMS system also needs to capture interdependencies in more detail than identified using TSP.

As stated previously, TSP breaks its major tasks down into granular tasks that can be completed within a one-week period. This can be considered too much detail to maintain in a consolidated project plan. So translation between a TSP task and an EVMS activity is done by translating SUMS assembly items to activities in EVMS. The resources are then assigned to the EVMS activities by determining the resources assigned to the corresponding TSP tasks.

The task hours planned for the SUMS assembly items must be converted into staff hours. When we first made this conversion, we did what any good planner does when he or she has no historical data: We estimated based on personal experience. We knew the average task hours per week the TSP team used for estimation, and we knew the average hours per week an engineer worked, thus giving us a starting ratio for task hours to staff hours. Once we collected some historical data, we determined that based on the type of work being performed, the task hour to staff hour ratio varies between 2.0 and 2.62. That is, it takes between 2.0 and 2.62 staff hours for every TSP task hour performed. Knowing this, we accurately converted task hours planned during a TSP launch to staff hours for EVMS purposes. Once the estimated staff hours were determined, we used the resource hourly rates to develop cost estimates.

The duration of EVMS activities can be determined by using the start and completion date of the first and last phase mapped to the TSP assembly. EVMS milestones are updated based on any milestones identified during the TSP launch, along with milestones identified from other project entities.

Using the task log in Table 1 (see next page) as an example, one of the activities in EVMS is called *Widget A*. Jack, John, and Jill are assigned resources for Widget A. According to the task log, Jack is assigned 200 planned task hours, and John and Jill are both assigned 44 planned task hours. Using the TSP task hour to staff hour ratio of 2.62, Jack would be assigned 524 (200*2.62) staff hours, and John and Jill would be assigned 115 (44*2.62) staff hours in EVMS for Widget A. The duration for Widget A in the EVMS would be 95 days because the planned start week and completion date is week 5 and week 24, respectively ((24 weeks minus 5 weeks)*5 days/week = 95 days). The predecessor of Widget A would be Widget B, and the successor of Widget A would be Widget C because of the order established during the TSP launch.

Once the conversion of TSP launch data to EVMS data is done, additional analysis needs to be made of the project's consolidated plan before it is considered complete. TSP off-task activities need to be identified and budgeted in the EVMS system. This includes TSP roles such as planning manager, test manager, design manager, and team lead. These tasks should be as specific as possible so that the correct method of EV can be assigned, trying to avoid LOE activities as much as possible so that project LOE does not exceed 10 percent of the total effort.

Dependencies between the TSP team's activities and other disciplines should be identified and linked in the EVMS. This

allows for the establishment of the project's critical path. The dependencies should also be analyzed to determine the impact to the other disciplines' plans, and to determine any resource allocations such as facilities' needs to be changed. Once this additional analysis is completed, the consolidated plan is ready to be baselined. The baseline is the basis in which the project's performance is measured against using EV.

At this point, you may be thinking it is a lot of work to baseline a consolidated project plan – and you would be correct. In fact, most large projects only rebaseline once or twice a year and only if the projects' overall performance against the baseline varies so much that the plan no longer reflects reality sufficiently to manage the project. So how do you take into account that TSP teams usually relaunch every three to four months?

This method required that the TSP team plan the entire effort up front, even if the effort is over a year in duration. When TSP teams relaunch, the current plan in the EVMS is updated but not the baseline. The only time the EVMS baseline will be updated with a team's relaunch data is when the entire project undergoes a project rebaseline.

During the period that the EVMS baseline does not reflect the TSP team's relaunch, the TSP relaunch data will be used as the basis for any EVMS action plans developed due to a deviation outside the project's defined acceptable variance, usually plus or minus 10 percent cost and schedule. In other words, a relaunch is simply a mechanism for developing a detailed action plan for correcting inaccurate plans developed during a TSP team's initial launch.

It is important to note that even though the overall team plan may extend for several years, the TSP's detailed plans extend for only a few months [3]. Thus, it should be expected that once the TSP team goes beyond the first few months following the TSP launch, the team will begin to vary beyond the EVMS baseline due to the fact that TSP launches and relaunches are not designed to develop a detailed plan beyond a three- to four-month period.

For example, Figure 1 represents the consolidated EV for a multi-disciplined project that consists of four TSP teams – system engineers, hardware engineers, domain experts, test engineers – and other support personnel. Figure 1 shows that both the cost and schedule are within a 10 percent variance, thus the project is performing within expected parameters.

Figure 2 is an EVMS report generated at the TSP Team A level of the multi-disciplined WBS. It shows that TSP Team A is behind schedule by 11 percent and over budget by 49 percent when comparing the team's current status against its baseline plan. Because this team is outside the 10 percent cost and schedule thresholds, the team is required to develop an action plan on how it will address these variances. One way for the team to address these variances is to conduct a relaunch, which would provide very detailed plans. Both the TSP team and project management can then use these detailed plans to mediate the impact to the overall project. Because the project is performing well, based on Figure 1, TSP Team A's performance would not justify the expense or effort needed to rebaseline the entire multi-discipline project. Thus, for EVMS purposes, TSP Team A would be tracked against its action plan.

Converting TSP EV Into EVMS EV

TSP teams can use a modified percent-complete method when converting TSP EV into EVMS EV. I mentioned earlier that percent complete could be subjective due to the lack of unbiased judgment. In this case, because TSP tasks are at a more granular level than the EVMS activities, the TSP tasks become the unbiased element used to determine percent complete.

One way of updating the completion of a TSP task is using the percent-complete method, which is assigned to the EVMS activity proportionally to the total number of unique TSP tasks mapped to the TSP assembly. For example, if five unique tasks are mapped to Assembly A and two of the tasks have been completed, then Activity A, in the EVMS, would be 40 percent complete. Some TSP tools such as Process Dashboard automatically calculate the percent complete at a given assembly level, in which case the number would be the percent complete in the EVMS for the corresponding activity.

Actual cost and actual staff hours expended on an activity in the EVMS should be recorded using a timecard sys-

Table 1: *TSP Task Log*

Assembly	Phase	Task	Resources	Estimated Size	Size Measure	Rate (per hour)	Estimated Hours	Engineers	Plan Hours	Plan Date	Plan Week
Widget B	PM	Widget B - Post-mortem	Jack	3	LOC	8.3	0.4	1.0	0.4	7/19/2004	5
Widget A	PLAN	Widget A - Research and Planning	Jack	2,000	LOC	200.0	10.0	1.0	10.0	7/19/2004	5
Widget A	DLD	Widget A - Detailed Design	Jack	2,000	LOC	47.6	42.0	1.0	42.0	8/23/2004	10
Widget A	TD	Widget A - Test Development	Jack	2,000	LOC	200.0	10.0	1.0	10.0	9/6/2004	12
Widget A	DLDR	Widget A - DLD Review	Jack	2,000	LOC	125.0	16.0	1.0	16.0	9/13/2004	13
Widget A	DLDINSP	Widget A - DLD Inspection	Jack, John, Jill	2,000	LOC	90.9	22.0	1.0	22.0	9/27/2004	15
Widget A	CODE	Widget A - Code	Jack	2,000	LOC	47.6	42.0	1.0	42.0	10/25/2004	19
Widget A	CR	Widget A - Code Review	Jack	2,000,	LOC	153.8	13.0	1.0	13.0	11/1/2004	20
Widget A	COMPILE	Widget A - Compile	Jack	2,000	LOC	400.0	5.0	1.0	5.0	11/1/2004	20
Widget A	CODEINSP	Widget A - Code Inspection	Jack, John, Jill	2,000,	LOC	90.9	22.0	1.0	22.0	11/15/2004	22
Widget A	UT	Widget A - Unit Test	Jack	2,000	LOC	125.0	16.0	1.0	16.0	11/29/2004	24
Widget A	PM	Widget A - Post-mortem	Jack	2,000	LOC	1000.0	2.0	1.0	2.0	11/29/2004	24
Widget C	PLAN	Widget C - Research and Planning	Jack	400	LOC	200.0	2.0	1.0	2.0	11/29/2004	24

Note: For DLDINSP and CODEINSP, the number of engineers is listed as 1.0. This is due to the replication of task that occurs using the Software Engineering Institute's TSP tool. This is only one of many equally valid methods used to address the replication of tasks to multiple individual workbooks when multiple engineers are assigned as a resource.

tem or other mechanism and not by using the task hour to staff hour ratio. This conversion should only be used for planning purposes. Both TSP and non-TSP teams need to collect these actual staff hours expended in the same manner to maintain consistency and validity of EVMS data.

Conclusion

TSP allows software teams to consistently meet commitments [4]. TSP provides an extraordinary amount of data that can be used for project planning and tracking. This data can be effectively incorporated into a consolidated multi-disciplined project plan. By incorporating the TSP data into an EVMS that tracks the entire project and not just the software development effort, the project is able to consider how work performance by one group impacts other groups within the project.

TSP encapsulates many traditional project planning and tracking elements such as WBS, tasks, and resource allocation. TSP does not fully encapsulate other concepts of traditional project planning and tracking such as critical path and dependencies.

TSP uses the 0/100 method of EV. Other EV methods not used by TSP include 50/50, percent complete, and LOE. It is important to select the appropriate method of EV to accurately measure the performance of an activity. When selecting an EV method for a project, duration and type of activity are the primary considerations.

Most of the information required to develop an EVMS project plan can be obtained from a TSP launch. In addition to this data, you must consider TSP off-task activities (LOE), dependencies between the TSP team's activities and other disciplines, and the project's critical path.

For project tracking purposes, once the TSP data has been incorporated into the EVMS project baseline, a TSP team can use a modified percent-complete method when converting TSP EV into EVMS EV. Because TSP tasks are at a more granular level then the EVMS activities, the TSP tasks become an unbiased element used to determine percent complete.

Most large projects only rebaseline the project plan once or twice a year, or when the project's overall performance against the baseline varies so much that the plan no longer reflects reality. Including the entire software effort in the EVMS baseline requires that the TSP team launch the entire effort – not just the next three- to four-month effort. For EVMS, a TSP relaunch is used simply as a mechanism for developing a detailed action plan for correcting inaccurate plans developed during a TSP team's initial launch or during the initial EVMS baseline.

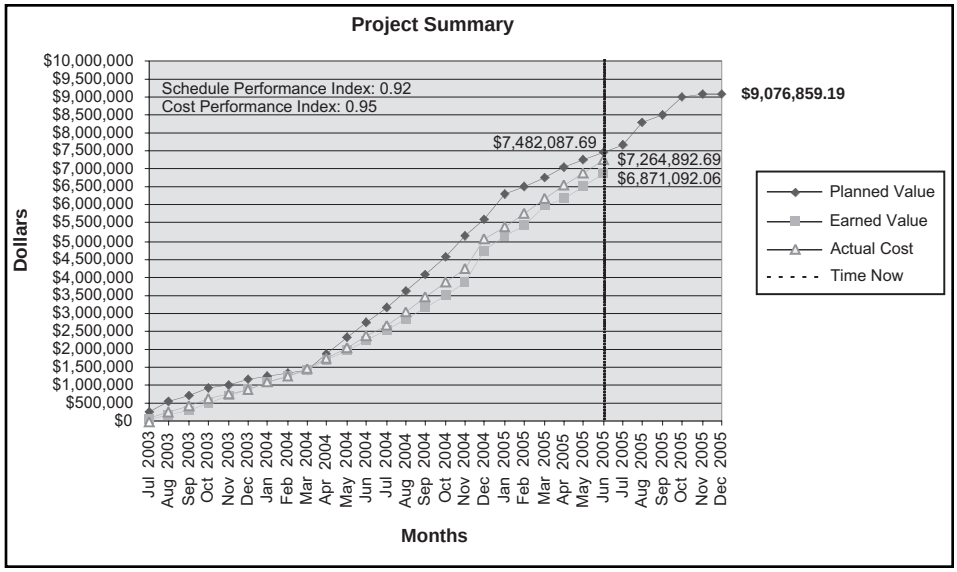


Figure 1: *Project Summary EVMS Report*

TSP does not eliminate the need for developing a consolidated project plan and tracking progress against the plan, especially in the world of very large and complex systems that consist of personnel from multiple disciplines such as software engineering, system engineering, hardware engineering, domain expertise, test engineering, and other support personnel.

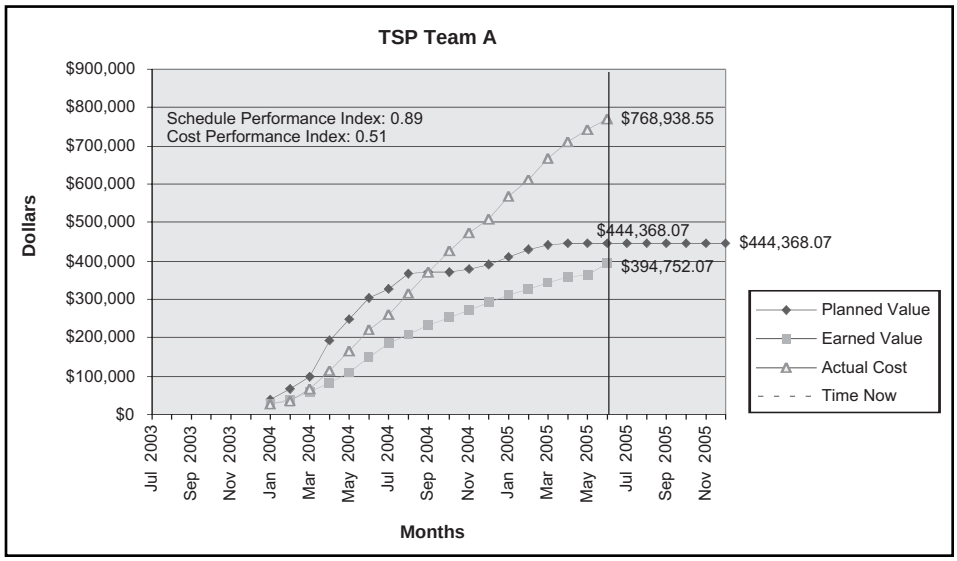
Acknowledgements

Many people have participated in the work that led to this article, but I would like to express a special thanks to Eileen Lang of Eagan McAllister Associates, Inc. Without her superior understanding of project planning and analysis, the integration of TSP into the Hawkeye: Navy Airborne Warning and Control System Aircraft's (E-2C HE2K) organizational EVMS would not have been possible.◆

References

1. Project Management Institute. A Guide to the Project Management Body of Knowledge (PMBOK Guide) Third Edition Excerpts. Newton Square, PA: PMI, Oct. 2004 <www.pmi.org/prod/groups/public/documents/info/pp_pmbokguidethirdexcerpts.pdf>.
2. Humphreys, Gary C. Project Management Using Earned Value. Orange, CA: Humphreys & Associates, Inc., 2002.
3. Humphrey, Watts S. PSP: A Self-Improvement Process for Software Engineers. Addison-Wesley, 2005.
4. Tuma, David, and David R. Webb. "Personal Earned Value: Why Projects Using the Team Software Process Consistently Meet Schedule Commitments." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2004/03/0503Tuma.html>.

Figure 2: *TSP Team A EVMS Report*



Coming Events

April 2-6
9th Communications and Networking Simulation Symposium
Huntsville, AL
www.scs.org/confernc/springsim/springsim06/cfp/cns06.htm

April 3-7
The 3rd International Conference on Software Process Improvement
Orlando, FL
www.icspi.com

April 10-12
3rd International Conference on Information Technology: New Generations
Las Vegas, NV
www.itng.info

April 19-21
Information Processing in Sensor Networks (IPSN 2006)
Nashville, TN
www.cs.virginia.edu/~ipsn06

April 19-21
19th Conference on Software Engineering Education and Training (CSEE&T 2006)
Oahu, HI
http://db-itm.cba.hawaii.edu/cseet2006/index.htm

April 24-28
2nd NASA/IEEE Systems and Software Week (SASW 2006)
Columbia, MD
www.systemsandsoftwareweek.org

May 1-4
2006 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

May 7-11
Computer Audit, Control, and Security Conference (CACS 2006)
Orlando, FL
www.isaca.org

About the Author

Timothy A. Chick is the software manager for the Hawkeye: Navy Airborne Warning and Control System Aircraft (E-2C HE2K) Software Support Activity for the Naval Air Warfare Center (NAVAIR), a division of the Department of the Navy at Patuxent River, Md. He is also the Software Engineering Process Group lead and a certified Personal Software Process℠ instructor and trained Team Software Process℠ coach. Chick has been with the E-2C program for more than six years. He has held several positions, including project team lead and software engineer. He has a Bachelor of Science in computer engineering from Clemson University and a Master of Science in computer science from Johns Hopkins University.

Naval Air Warfare Center – Aircraft Division
BLDG 2185 STE 1190-B2 4.1.4
22347 Cedar Point RD UNIT 6
Patuxent River, MD 20670
Phone: (301) 342-0489
Fax: (301) 757-3219
E-mail: timothy.chick@navy.mil

Web Sites

Software Engineering Institute
www.sei.cmu.edu
The Software Engineering Institute (SEI℠) is a federally funded research and development center sponsored by the Department of Defense to provide leadership in advancing the state of the practice of software engineering to improve the quality of systems that depend on software. SEI helps organizations and individuals improve their software engineering management practices.

Project Management Institute
www.pmi.org
Established in 1969, the Project Management Institute (PMI) is a not-for-profit, project-management professional association with more than 100,000 members in 125 countries. PMI members are in many different industry areas, including aerospace, automotive, business management, construction, engineering, financial services, information technology, pharmaceuticals, and telecommunications. PMI publishes "A Guide to the Project Management Body of Knowledge," and its Project Management Professional certification is the world's most recognized professional credential for individuals associated with project management. In 1999, PMI became the first organization in the world to have its certification program attain International Organization for Standardization 9001 recognition.

Software Program Managers Network
www.spmn.com
The Software Program Managers Network (SPMN) is sponsored by the deputy under secretary of defense for Science and Technology, Software Intensive Systems Directorate. It seeks out proven industry and government software best practices and conveys them to managers of large-scale Department of Defense software-intensive acquisition programs. The SPMN provides consulting, on-site program assessments, project risk assessments, software tools, guidebooks, and specialized hands-on training.

INCOSE
www.incose.org
The International Council on Systems Engineering (INCOSE) was formed to develop, nurture, and enhance the interdisciplinary approach to enable the realization of successful systems. INCOSE works with industry, academia, and government in these ways: provides a focal point for disseminating systems engineering knowledge, promotes collaboration in systems engineering education and research, assures the establishment of professional standards for integrity in the practice of systems engineering, and encourages governmental and industrial support for research and educational programs to improve the systems engineering process and its practices.

Factors Affecting Personal Software Quality

Dr. Mark C. Paulk
Carnegie Mellon University

Understanding the factors that influence software quality is crucial to the continuing maturation of the software industry. An improved understanding of software quality drivers will help software engineers and managers make more informed decisions in controlling and improving the software process. Data from the Personal Software Process℠ provides insight into interpersonal differences between competent professionals as increasingly disciplined processes are adopted. Program size, (empirically measured) programmer ability, and disciplined processes significantly affect software quality. Factors frequently used as surrogates for programmer ability, e.g., years of experience, and technology, e.g., programming language, do not significantly impact software quality, although they may affect other important software attributes such as productivity. An understanding of these factors may help managers implement practices that support high-quality software.

The research reported in this article was part of a larger investigation into the relationship between process discipline and software quality [1]. It focuses on the product, technology, and programmer ability factors that may affect software quality in addition to the process factors. My research confirms that a disciplined process affects software quality. It also confirms that programmer ability affects software quality and, more importantly, shows that even top performers can improve their performance by a factor of about 2X by following disciplined processes. Commonly used surrogates for ability such as seniority and academic credentials are, however, likely to be ineffective measures.

The research uses data from the Personal Software Process℠ (PSP℠), which applies process discipline to the work of the individual software professional in a classroom setting. PSP is taught as a one-semester university course at several universities or as a multi-week industry training course. It typically involves 10 programming assignments, using increasingly sophisticated processes [2]. The life-cycle processes for PSP are planning, designing, coding, compiling, testing, and a post-mortem activity for learning. The primary development processes are designing and coding, since there is no requirements analysis step.

When discussing *quality* in the software industry, *defects* are the common indicator, although software quality characteristics include functionality, reliability, usability, efficiency, maintainability, and portability. Although other aspects of quality are important, software quality is measured as *defect density in testing* in the analyses described in this article.

Potential explanatory variables identified in prior research can be divided into categories related to the product and application domain, the technologies used, the software engineering processes followed, and the ability of the individuals doing the work. Quality issues related to the customer requirements are outside the scope of this research. Although requirements volatility is a significant quality concern in software projects, requirements volatility is not an issue for PSP.

Many explanatory factors for software quality from models such as COQUALMO [3] are out of scope for this research because they address project and team issues that are not relevant to the PSP environment. This highlights the challenges in generalizing PSP results to software work in general, but factors important for individual performance should also be important for teams, projects, and organizations.

> *"My research confirms ... that programmer ability affects software quality and, more importantly, shows that even top performers can improve their performance by a factor of about 2X by following disciplined processes."*

There are four PSP major processes – PSP0 to PSP3 – with minor variants for the first three (PSP3 can also be considered a minor extension of PSP2). Each level builds on the prior level by adding a few engineering or management activities. This minimizes the impact of process change on the engineer, who needs only to adapt the new techniques into an existing baseline of practices. Design and code reviews are introduced in assignment No. 7. Design and code reviews are personal reviews conducted by an engineer on his or her own design or code. They are intended to help engineers achieve 100 percent yield: All defects are removed before compiling the program. Design templates are introduced in assignment No. 9. The design templates are for functional specifications, state specifications, logic specifications, and operational scenarios.

PSP students are asked to measure and record three basic types of data: time (effort), defects, and size (lines of code [LOC]). All other PSP measures are derived from these three basic measures.

Data analyzed in this research included the data used in the Hayes and Over study [4], as well as additional data collected through 2001. These rich data sets allow sophisticated statistical analyses ranging from simple regression models to multiple regression models to mixed models that incorporate random effects and repeated measures. The conclusions reported are based on consistent results for multiple data sets. The data sets are usually split by assignment 9A or 10A (to remove potential differences in problem complexity and ensure a relatively mature process) and by programming language (to remove technology differences), both including and excluding outliers. A data set could therefore be described as (9A, C, No Outliers). In some cases, e.g., when investigating the impact of programming language used, different data splits are used as appropriate.

Confirming PSP Quality Trends

Process maturity is a generic concept that

℠ Personal Software Process and PSP are service marks of Carnegie Mellon University.

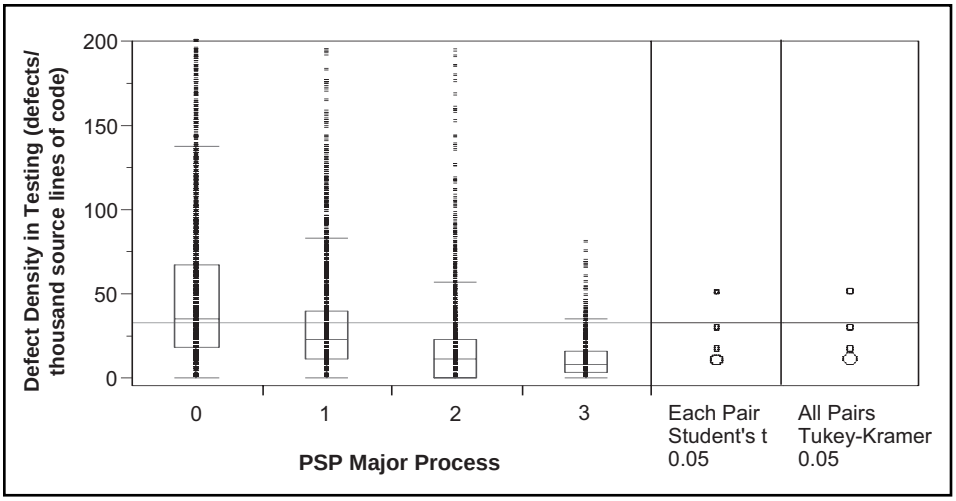


Figure 1: *Trends in Software Quality*

can be considered *low* for PSP0 and *high* for PSP3. It is interesting to note that in COQUALMO, *process maturity* has the highest impact of all factors on defect injection. As can be observed in Figure 1, which shows the differences in *defect density in testing* for each PSP *major process*, the software quality trend across the PSP's major processes is apparent, confirming prior analyses.

The comparison circles in the right part of Figure 1 indicate that the means for each of the *PSP major processes* are significantly different. The *Each Pair*, *Student's t*, and the *All Pairs*, *Tukey-Kramer honest significant difference* are separate tests of the hypothesis that the *defect density in testing* for the *PSP major processes* is significantly different. When there is no overlap of the comparison circles, as in this case, there is a significant difference, and we can conclude that the *defect density in testing* for *PSP0 > PSP1 > PSP2 > PSP3*. The differences between the PSP major processes are both statistically and practically significant. Differences greater than 4X show a decrease in *defect density in testing* of more than 75 percent over the course of PSP. This is a quality improvement that would be of interest and value to most software professionals.

These data include some outliers. Excluding outliers without causal analysis, as well as including them when they are atypical, can skew results. The analyses reported in this article are performed both with and without outliers, using inter-quartile limits (limits set at 1.5 times the inter-quartile range beyond the 25 percent and 75 percent quartiles) to identify outliers.

Program Size

Problem complexity would not appear to be a significant factor in PSP, given the relative simplicity of the assignments; programs smaller than 10,000 LOC are usually considered simple programs. S*olution complexity* is measured by new and changed LOC. A student's preferred style in optimizing memory space, speed, reliability (e.g., exception handlers), generality, reuse, etc., can lead to radically different solution complexities. Since PSP does not impose performance requirements, students have significant latitude in how they solve the problems – latitude that is available in many industry contexts as well. Using defect density as a quality measure should normalize these differences in solution complexity.

Since defect prediction models typically use *program size* as a predictor variable (and many use it as the only predictor variable), this variable is expected to be significant. *Program size* was shown to be a statistically significant effect on *defect density in testing* in all of the four data sets analyzed including outliers, but in only two of the four data sets excluding outliers. The preponderance of the evidence indicates that *program size* is related to software quality, although more weakly for PSP than for many prior defect prediction models. These somewhat atypical results highlight the impact of individual differences on software quality, although individual differences may be smoothed out by team performance on a project.

Programmer Ability

Programmer/analyst capability is difficult to objectively determine, but the PSP data provide a direct and objective measure of programmer ability. This is illustrated in Figure 2, where *defect density in testing* was averaged for the first three assignments and used to identify the top, middle two, and bottom quartiles for student performance. Note that using defect density as the measure for ability means that decreasing value of the measure corresponds to increasing ability, which means that the *top quartile* of programmers is at the bottom of the graph. *Programmer ability*, as measured by this variable, was shown to be statistically significant for every data set, including and excluding outliers.

The students who were the top-quartile performers on the first three assignments tended to remain top performers (with the smallest *defect density in testing*) in the later assignments, the middle performers tended to remain in the middle, and the bottom-quartile performers (with the largest *defect density in testing*) tended to remain at the bottom. (A spike in the data for assignment No. 3 is consistently observed for all measures, suggesting that assignment No. 3 is somewhat more complex than the norm for the PSP assignments.)

While top-quartile students performed better than those in the bottom quartile on average, a disciplined process leads to significantly better performance for the bottom-quartile students, and even the top-quartile students improved markedly. Throughout the PSP course, top-quartile students improved their software quality by a factor of more than two, and bottom-quartile students improved theirs by a factor of more than four. Variation in performance within each quartile also decreased markedly.

Potentially Confounding Variables

A number of variables could affect the analyses if not appropriately addressed. Potential confounding variables include those associated with instructor differences; surrogates for ability such as credentials, experience, recent experience, and breadth of experience; and technology factors such as the programming language used.

PSP Classes

There are two situations that could cause systemic differences across PSP classes: (1) changes in the teaching materials used in the PSP class, or (2) differences between instructors. The possibility of a trend due to systemic changes in the student population does not appear likely since there are no known reasons for a systemic change in the student population for PSP, although sporadic cases of exceptionally poorly prepared or well-prepared classes could occur. The PSP class has been based on the text "A

Discipline for Software Engineering" since its publication in 1995; prior classes used drafts of the manuscript. All PSP instructors are qualified and authorized by the Software Engineering Institute, and instruction is typically done by teams of instructors. There would not appear to be any reason for changes in PSP class performance over time although there might be cases where particular classes might have significantly different results because of special causes of variation.

Combining a small number of finishing students with one or two students struggling to finish assignment No. 9 and/or No. 10 leads to the occasional atypical class, which is not unexpected given the large number of classes being analyzed. There does not appear to be a statistically or practically significant trend over time, and it seems reasonable to conclude that, in general, PSP classes are relatively stable learning environments, although some students may have trouble on some assignments.

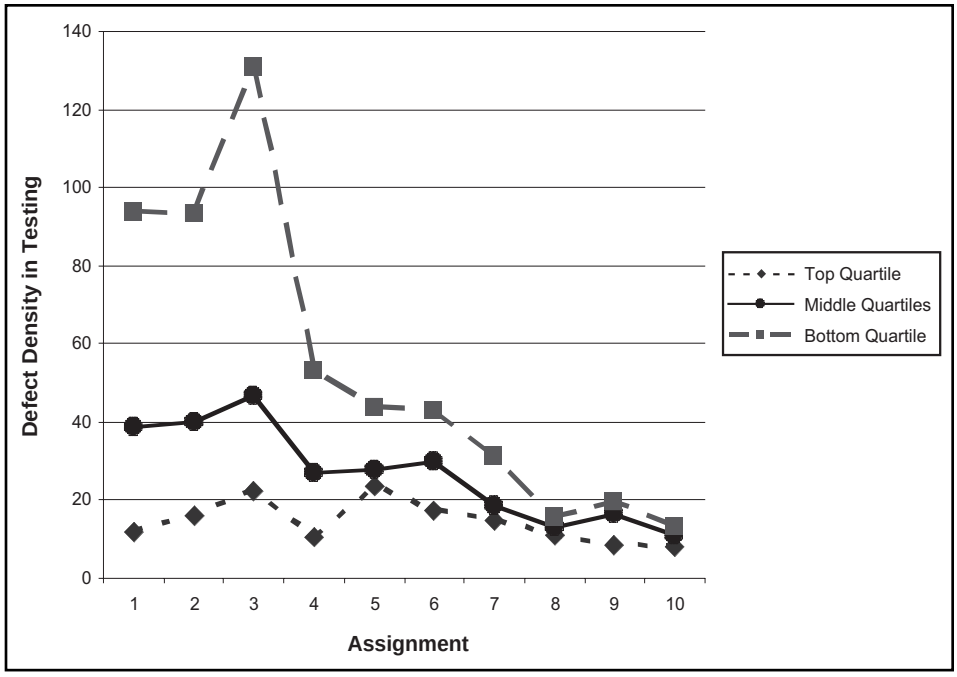


Figure 2: *Trends for Programmer Quartiles*

Finishing the Course (Or Not)

If there is a difference in performance on the early assignments between the people finishing the course and those who do not, the student population may be different from the general programming population. Finishing the course was shown to have a statistically significant effect on *defect density in testing* in only one of the four data sets analyzed, including outliers, and in none of the data sets excluding outliers. This suggests that people finishing the PSP course are reasonably typical of programmers who choose to take the PSP course. This does not, however, necessarily indicate that PSP students are representative of the population of software professionals in general.

Highest Degree Attained

As illustrated in Figure 3, the *highest degree attained* (doctorate [Ph.D.], Master of Science [MS], Bachelor of Science [BS], or Bachelor of Engineering [BE]) was not shown to be a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers. The overlap in both sets of comparison circles graphically show that *defect density in testing* for students with a doctorate overlaps that of students with a bachelor's or master's degree.

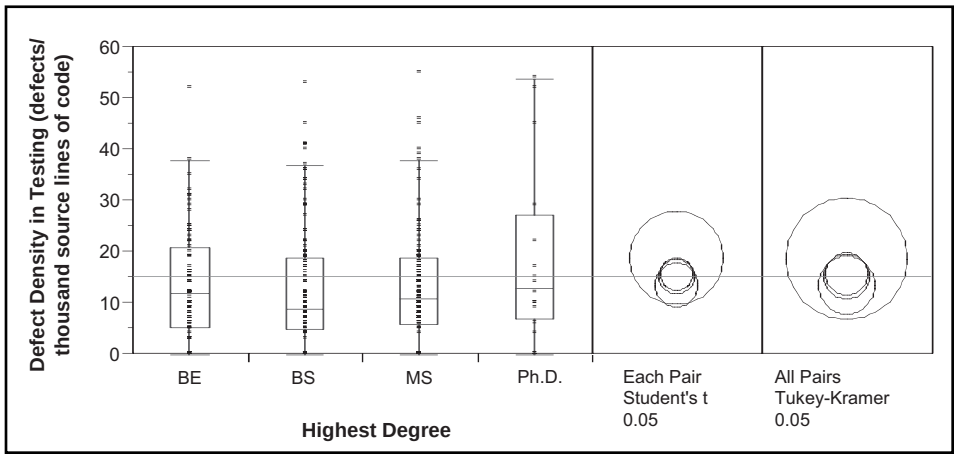
This result indicates nothing about whether people who pursue additional academic credentials will improve their ability; it simply indicates that they are not useful distinguishers between students in PSP. Although degree credentials are not a significant factor for PSP assignments, educational credentials may contribute to improved performance for more complex programs where deeper, more extensive domain or engineering knowledge may be crucial to understanding both the problem and potential solutions.

Years of Experience

Although some researchers have empirically found that *years of experience* are related to quality, it was not shown to have a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers. When the studies finding that experience was a significant factor were performed in the 1970s and 1980s, entry-level programmers were relatively unfamiliar with computers. The familiarity of the general population with computers has grown markedly over the last few decades. Students and entry-level programmers in the 1990s were likely to be well acquainted with computers before beginning their professional careers. The consequence is that the operational definition of *years of experience* for programmers is likely to have shifted in the last three decades. The results of the earlier studies may have been valid and yet be irrelevant to today's population of programmers.

This does not imply that experience does not affect ability. Holmes found that for his PSP data, collected over a seven-year period, developer experience was a significant factor [5]. His results, however, apply to a single individual engaged in applying PSP as part of a systematic improvement program. They indicate that individual professionals can substantially improve their performance over time, but they cannot be generalized

Figure 3: *Differences for Highest Degree Attained*



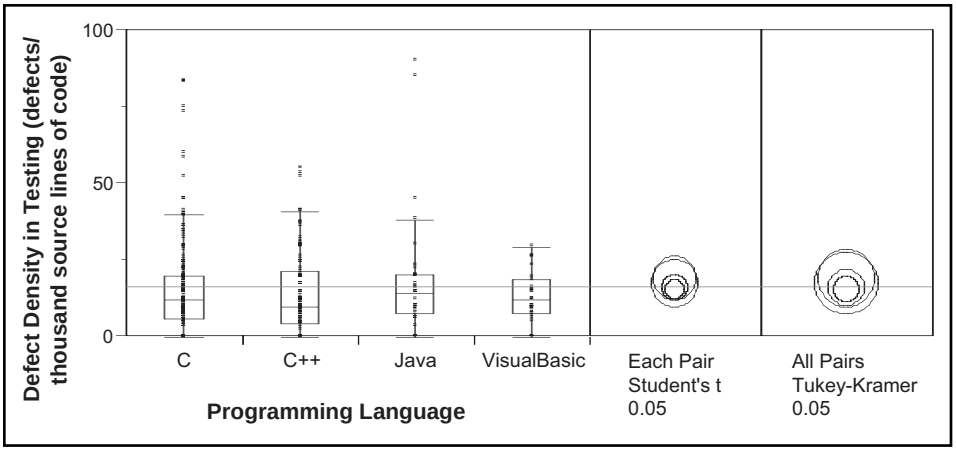


Figure 4: *Differences Between Programming Language Used*

across different individuals, which is the point of this analysis.

Industry studies of the effect of *years of experience* on quality typically use the average number of years for the team [6]. In averaging experience across the team, related factors in assigning professionals to the team with diverse backgrounds may impact the operational meaning of *experience*. The relevant factors may be related to a diverse team with a variety of skills and capabilities – *years of experience* may be confounded with other factors related to the skills of the team.

Number of Languages Known

The *number of languages known* may suggest the diversity of experience of the programmer, which in turn may indicate ability. While length of experience is usually considered a poor surrogate for ability, breadth of experience is considered a more realistic indicator [7], particularly for programmers with less than three years of experience [8]. The number of languages that a programmer has a working knowledge of may be considered a reasonable surrogate for breadth of experience. The *number of languages known* was shown to have a statistically significant effect on defect density in testing in only one of the four data sets analyzed, including outliers, and in none of the data sets excluding outliers.

Percent of Time Programming in the Previous Year

Recent experience in programming is primarily a concern for learning curve effects associated with programming skills in general. Given the small size of the PSP assignments, it seems likely that the bulk of any learning curve effects associated with basic programming skills are concentrated in the first few assignments. *Percent of time programming in the previous year* was not shown to have a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers.

Programming Language

Although some researchers have noted that the *programming language* used does not appear to be significantly correlated with productivity or quality except at a gross level [9], others have found that programming language can have a significant impact on both. For example, in defining the backfiring technique for estimating function points based on LOC, Jones found noticeable differences between different languages: It takes 128 LOC in C to implement one function point, but only 53 in C++ [10]. As illustrated by the overlap in the comparison circles in Figure 4, the *programming language* used was not shown to have a statistically significant effect on *defect density in testing* for any of the data sets analyzed, excluding or including outliers.

Although language may not significantly affect defect density, the productivity difference between languages implies that the number of defects will vary significantly depending on the language(s) used. In other words, if it takes twice as many LOC to implement a program in language A as in language B, then there will be twice as many opportunities for defects in the language A program as for the language B program, even if the defect density is the same. Because of the impact of programming language on productivity, and the related effects on process variables such as review rates, the *programming language* used should not be ignored in analyzing software quality, although it may not be a statistically significant variable for quality (as measured by *defect density in testing*) when considered in isolation.

Conclusion

This research found that (1) process discipline is an important factor for software quality; (2) *program size*, which is an indicator of solution complexity, is an important quality factor; and (3) *programmer ability* is an important quality factor when empirically measured. Other variables that may appear to be plausible surrogates for important areas such as ability and technology were not shown to be significant.

The issue of relative ability of programmers is particularly important, since the finding that even top-quartile performers improve more than 2X refutes those who resist the need for discipline, while acknowledging that their performance is superior and their opportunities for improvement are less than many of their colleagues.

This research supports the premise of PSP and similar process improvement strategies: Disciplined software processes result in superior performance compared to *ad hoc* processes. This improvement can be seen in both improved performance and decreased variation. It can be inferred that this is the minimum level of improvement that can be expected for a set of programmers since continual improvement can be expected after the PSP class.

The practical implications of this research for software managers and professionals are relatively simple, although they may be challenging to address. First, although *programmer ability* is a crucial factor affecting software quality, surrogates such as seniority and academic credentials are inadequate for ranking programmers, and empirical measures that are more effective may cause dysfunctional behavior if used for determining raises and promotions [11]. Second, consistent performance of recommended engineering practices improves software quality, even for top performers, who may resist discipline and measurement-based decisions because of their superior performance.◆

References

1. Paulk, M.C. "An Empirical Study of Process Discipline and Software Quality." Ph.D. diss., University of Pittsburgh, 2005 <http://etd.library.pitt.edu/ETD/available/etd-07082004-155917>.
2. Humphrey, W.S. A Discipline for Software Engineering. Reading, MA: Addison-Wesley, 1995.
3. Boehm, B., C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D. Reifer, and B. Steece. Software Cost Estimation With COCOMO II. Upper Saddle River,

NJ: Prentice Hall, 2000.
4. Hayes, W. and J.W. Over. "The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers." Pittsburgh, PA: Software Engineering Institute, 1997.
5. Holmes, J.S. "Optimizing the Software Life Cycle." ASQ Software Quality Professional 5.4 (Sept. 2003): 14-23.
6. Zhang, X. "Software Reliability and Cost Models with Environmental Factors." Ph.D. diss., Rutgers, 1999.
7. Curtis, B. "The Impact of Individual Differences in Programmers." Working With Computers: Theory Versus Outcome. Ed. G.C. van der Veer. London: Academic Press, 1988: 279-294.
8. Sheppard, S.B., B. Curtis, P. Milliman, and T. Love. "Modern Coding Practices and Programmer Performance." IEEE Computer 12.12 (Dec. 1979): 41-49.
9. DeMarco, T., and T. Lister. Peopleware. 2nd ed. New York: Dorset House, 1999.
10. Jones, C. "Backfiring or Converting Lines of Code Metrics Into Function Points." Burlington, MA: Software Productivity Research, Oct. 1995.
11. Austin, R.D. Measuring and Managing Performance in Organizations. New York: Dorset House Publishing, 1996.

About the Author

Mark C. Paulk, Ph.D., is a senior systems scientist at the Information Technology Services Qualification Center at Carnegie Mellon University. From 1987 to 2002, Paulk was with the Software Engineering Institute at Carnegie Mellon, where he led the work on the Capability Maturity Model® for Software. Prior to joining Carnegie Mellon, he was a senior systems analyst for System Development Corporation at the Ballistic Missile Defense Advanced Research Center in Huntsville, Ala. He is a Senior Member of the Institute of Electrical and Electronics Engineers and a Senior Member of the American Society for Quality. Paulk has a Bachelor of Science in mathematics and computer science from the University of Alabama in Huntsville, a Master of Science in computer science from Vanderbilt University, and a doctorate in industrial engineering from the University of Pittsburgh.

IT Services Qualification Center
Carnegie Mellon University
Pittsburgh, PA 15213
Phone: (412) 268-5176
E-mail: mcp@cs.cmu.edu

CALL FOR ARTICLES

Do You Know How to Merge Hardware and Software?

CROSSTALK is looking for an introductory article that discusses merging hardware and software. Since most of CROSSTALK's articles come from our readers, we know this is our best way to reach you with this request. Call CROSSTALK Associate Publisher Beth Starrett at (801) 775-4158 or e-mail <beth.starrett@hill.af.mil> to discuss your experience in merging hardware and software, and how to transform your knowledge into an article that can get your ideas out to more than 100,000 CROSSTALK readers.

Also, take a look at CROSSTALK's new Theme Calendar format at <www.stsc.hill.af.mil/crosstalk/theme.html>. We now provide a link to each monthly theme, giving greater detail on the types of articles we're looking for.

Designing in UML With the Team Software Process

David R. Webb, Ilya Lipkin, and Evgeniy Samurin-Shraer
309 Software Maintenance Group

The Team Software Process℠ (TSP℠) is a good project management tool that enforces a disciplined approach to software engineering, drastically improving cost and schedule performance and the production of quality products. One of the ways TSP improves product quality is through emphasis on design. A heavy design emphasis is also the hallmark of the newer programming environments. This article examines how modern design techniques can be used on a TSP project.

In late 2004, the 309 Software Maintenance Group (309 SMXG) at Hill Air Force Base took on a series of new projects. One of these projects focused on updating software in an embedded weapon system, a task that 309 SMXG was proficient in performing; however, this new system had been developed using a modern Unified Modeling Language (UML) auto code generation toolset, with which the group had no prior experience. Assessed as a Capability Maturity Model® (CMM®) Level 5 organization in 1998, and focusing on a CMM Integration℠ (CMMI®) assessment in 2006, the 309 SMXG was confident in its ability to deliver software on time and within budget for traditional projects, but determined that this newer system required a different approach.

With this in mind, 309 SMXG brought in an entirely new team of UML developers to work on the maintenance of the new weapon system. The obvious drawback to this approach was that this new team had little or no experience using the disciplined software engineering techniques required by the CMM and CMMI. Internal group policies, which the team was required to follow, demanded the project to tailor processes from the organizational standard and to follow the CMMI's specific practices for everything from project planning to quantitative project management. However, due to the inexperience of the team members, the project was struggling to come up to speed on these practices on a very short schedule.

Having had success in the past using the Software Engineering Institute's (SEI℠'s) Team Software Process℠ (TSP℠) [1] to bolster the group's own internal processes for newer teams, the 309 SMXG decided to use this approach on the new project. The resulting marriage between UML and TSP created a flexible and mobile design tool with a rigid and disciplined process.

TSP and CMMI

Simply put, TSP is CMM/CMMI Level 5 at a project level. It is supported by team members who perform Level 5 practices at a personal level using SEI's Personal Software Process℠ (PSP℠). A recent report by the SEI indicates that adopting the TSP will satisfy most of the specific practices of the CMMI process areas [2]. To quickly bring the team up to speed on the CMMI, the 309 SMXG put the entire software team through PSP training, which required about six weeks. They also trained the team in using a PSP tracking tool called the Process Dashboard that automates many of the personal planning and tracking activities required by the PSP.

> *"The resulting marriage between UML and TSP created a flexible and mobile design tool with a rigid and disciplined process."*

At the conclusion of this training, a certified SEI TSP launch coach took the team through a one-week TSP launch session. These sessions are used to determine stakeholder goals, establish team roles and processes, and produce detailed earned value, quality, and risk management plans. Since these are all key elements of the CMMI, the launch sessions are vital to the disciplined software engineering practices required by the model.

It was during this launch that the team encountered its first issues with the project's UML environment. To create the detailed earned value plans required by the TSP, each of the team's major tasks required some type of size criteria of task development for estimating purposes. In other words, the team needed a way of determining which tasks were larger than others and calculating how much effort those tasks would require. In a traditional software environment, the team would have estimated using source lines of code (SLOC) and converted from SLOC to effort using a team productivity rate. There were two problems with this traditional approach: (1) since the team was new, there was a lack of historical data upon which to base productivity estimates, and (2) the auto-generated code features of the UML environment made traditional size estimation very problematic. Since each of the team members had measured their effort and lines of code in the PSP class, the issue of productivity could have been addressed by using classroom averages of SLOC/hour; however, the problem of size estimation proved much more difficult.

UML Design With TSP

UML is a *language* developed by Grady Booch, James Rumbaugh, and Ivar Jacobson that uses object-oriented concepts and methodologies to model software systems [3]. Simply stated, UML consists of a set of diagrams that allow designers to examine a software program from several different points of

Table 1: *PSP Design Template Structure (SEI)*

Object Specification	Internal	External
Static	Logical Specification Template	Functional Specification Template
Dynamic	State Specification Template	Operational Scenario Template

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
℠ CMM Integration and SEI are service marks of Carnegie Mellon University.

view prior to creating the code [4]. The standard set of UML diagrams is the following:

- Use Cases.
- Class Diagrams.
- Object Diagrams.
- Sequence Diagrams.
- Collaboration Diagrams.
- State-Chart Diagrams.
- Activity Diagrams.
- Component Diagrams.
- Deployment Diagrams.

Combined views of these diagrams create a complete description of the software design.

As it turns out, the UML view of design does not differ significantly from the basic design techniques taught during the PSP course. In fact, the PSP design templates and design scripts provide a clear and concise description of steps needed to produce an effective design in UML. The PSP design-first technique uses four *orthogonal* views of any software design: internal static (module or part construction such as the logical layout of a module), internal dynamic (characteristic based upon changing values within the module), external static (the relationship of a module or part to other parts in the product), and external dynamic (the interactions this part or module has with other parts in the product). Each design view has a template to capture the information (see Table 1). The following is an example of how to use them [5].

Let us assume we want to develop software for a standard traffic light. The traffic light has three possible conditions: red (stop), yellow (caution), and green (go). The Operational Scenario Template (external/dynamic view) is used to capture the fact that the light consistently changes from green to yellow to red based solely upon a timed sequence. The State Specification Template (internal/dynamic view) captures the fact that there is a defined set of states through which the traffic light moves: green, to yellow, to red, and back to green again. It is not possible to go from green to red or yellow to green (see Table 2).

Now that requirements for the traffic light have been presented, it is time to capture them in the design. To perform this task, it is best to use a set of templates on the requirements; several templates will be used.

1. Operational Scenario Template (dynamic/external) is the system requirements, which are treated as use cases for the traffic light.

State #, Name	Description	Attributes
Initial	System not running	Timer event set to 30 seconds
Green	On timer time out event go to(goYellow) Yellow	Timer event set to 45 seconds
Yellow	On timer time out event go to(goRed) Red	Timer event set to 10 seconds
Red	On timer time out event go to(goGreen) Green	Timer event set to 30 seconds

Table 2: *Example of the State Specification Template (SEI)*

Object Specification	Internal	External
Static	Class Diagrams	Component Diagrams Deployment Diagrams Object Diagrams
Dynamic	Activity Diagrams State-Chart Diagrams	Use Cases Sequence Diagrams Collaboration Diagrams

Table 3: *PSP Design Structure for UML (SEI)*

2. Functional Specification Template (static/external) is used to describe the traffic light timer functionality and how it is used.

> *"... the PSP design templates and design scripts provide a clear and concise description of steps needed to produce an effective design in UML."*

3. State Specification Template (dynamic/internal) is used to capture the flow of events between states that are now colors of the traffic light (see Table 2).
4. Logic Specification Template (static/internal) can then be used to capture steps in pseudo code for the user-entered, action-code portion of UML. This template makes even the traditional process of coding almost trivial. For this example, there is no pseudo code needed as it is done entirely in UML events.

When all of these templates are completed, it becomes obvious how the logic must be constructed for the software to run the traffic light system design. It is now a simple task to draw the design in UML. The UML design then can auto-generate code at this stage (Figure 1).

Final UML designs produced, as shown in Table 3, fit into the same four quadrants as the PSP design templates. In Figure 1, the design description of the traffic light is captured using UML techniques. The PSP template used to implement this design description is the State Specification template, which is equivalent to the UML State-Chart Diagram.

The only other diagram created and required to complete this design is from

Figure 1: *Example of the Working UML Design Solution for Traffic Light State-Chart Diagram*

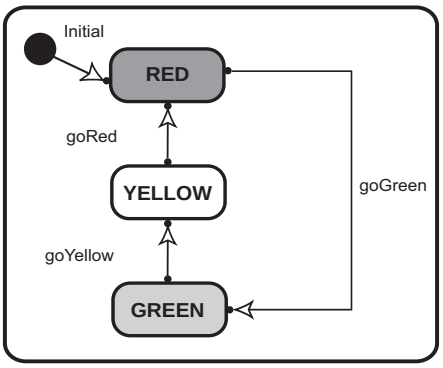
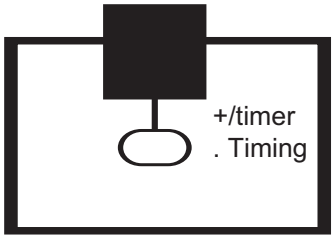


Figure 2: *Example of the Working UML Design Solution for Traffic Light Collaboration (Structure) Diagram*



the Functional Specification Template. This template describes timer functionality to the traffic light system, which translates into a collaboration diagram (Figure 2).

Using UML does not contradict the TSP philosophy, but in fact works quite nicely with this highly disciplined process. In addition, some UML tools (such as the IBM Rational Rose Real Time suite, used by the 309 SMXG project) offer the unique ability to utilize these diagrams to create auto-generated code. The chief reason that size estimation was so difficult for this new 309 SMXG project was due to auto-generation capability of UML. This seemed to be the only real issue in dealing with a modern UML design and development environment on a TSP project.

A New Size Metric

The problem with using a typical LOC counter to determine size was the auto-generated code. When drawing a single new line in a diagram (what you might think of as a single SLOC change), could (and often did) generate dozens of new and changed SLOC. This was due to the fact that the tool would *rethink* the entire software module, rather than just the single function being addressed. The result was that there existed little or no correlation between SLOC generated and effort required to produce the change. A traditional SLOC estimate would mean little or nothing to this team; something else was needed for size and effort estimation.

The size metric required for this project had to meet two criteria to be usable for estimating and tracking: (1) it had to correlate to the work performed, and (2) it had to be automatically measurable – counting any measurement by hand was deemed to be too slow and inaccurate. After a great deal of research, the 309 SMXG team decided to try out a rough approximation of engineer-generated and auto-generated SLOC. Using what limited data they had, the team's design manager determined that in an average build, approximately one-third of the code was *user developed* (i.e., directly correlated to the work performed). The remaining two-thirds was auto-generated with considerably less effort on the engineer's part. With this in mind, the team examined modules created by the original development team prior to the maintenance phase and applied these findings to determine *adjusted SLOC*. Modules were then estimated using Adjusted SLOC and the estimates converted to effort using PSP classroom productivity rates [6].

Once these metrics were determined, sizes were estimated and effort computed; this led to the production of the detailed earned value plan that is the hallmark of TSP projects.

Running the TSP Project

Since the metrics used to estimate and track the project size and effort were new, there was obviously some concern about the accuracy of the estimates. The TSP practice of tracking progress at the personal level and then *rolling* that data up to the team level allowed this project to find and correct potential estimate issues before the project missed any schedule deadlines. Effectively, the TSP and the use of the Process Dashboard kept members of the team on task and on schedule [7].

The process steps needed to complete the design were captured as earned value tasks on each individual software engineer's dashboard. The tasks were then broken down further into something that could be accomplished on a weekly basis. During product execution,

Table 4: *High Level Design Process Tasks*

Entry	Task	Exit

• Need to create or modify system design
• Project plans approved
• Resources identified and available
• Customer requirements configured
	• Research
	  • Gather data/documentation
	  • Identify assumptions
	• Analyze
	  • Analyze/functionally decompose requirements
	  • Perform risk analysis
	  • Identify required system resources (throughput, speed, memory, etc.)
	• Consider alternate solutions
	  • List the make/buy/reuse alternatives
	  • List other alternatives
	  • Utilize decision analysis and resolution as required
	  • Select solution(s)
	• Determine/design interfaces
	• Create design document(s)
	• Determine and document test requirements
	• Perform personal review
	• Hold peer reviews/perform corrective actions
	• Configure outputs in accordance with the project's configuration management plan
	• Conduct preliminary design review (PDR)
	  • Coordinate with relevant stakeholder
	Verification and Validation
	• Perform personal review
	• Hold peer reviews/perform corrective actions
	• Acceptance of preliminary design by the interdisciplinary team and the customer
	• Relevant stakeholders participate in creation of outputs and accept the preliminary design
	• Design simulation
	• PDR
		• Preliminary design completed
		• PDR completed
		• Record project data (effort, schedule, risk, defect data, minutes, lessons learned, etc.)

the team met weekly to ascertain both team and individual earned value. If a certain task was taking longer than expected, it became highly visible to the team members and the project manager during the weekly meeting. Any problem tasks were discussed during weekly team meetings and addressed by either reassigning it to another member or by addressing the scope with the customer early enough so as to not cause a schedule slip. In addition, these tasks were used to determine if some aspects of the design had been underestimated or overestimated for the development of the software. TSP allowed the team to catch problems at the first sign of an anomaly and to address it quickly, without significant cost to the customer or the project [7].

Furthermore, these tasks were based simply on the steps defined in the project that were tailored from the organizational process required by CMMI (see Table 4). The process descriptions (called *scripts* in the TSP) served as handy references for the team members, detailing what needed to be followed and completed on the dashboard tracking tool. The breakdown of the processes through the dashboard allowed for the design steps to be tracked individually.

One issue the team did uncover was a lack of dependency tracking. For example, if Team Member A needed Task 1 to be completed by Team Member B prior to working his/her task, then Team Member A's schedule is dependent upon Team Member B; however, this dependency is not reflected in the earned value plan created by the team during the initial launch.

This is due to the fact that the TSP earned value plan does not identify these dependencies. The team found that this is both a weakness and strength of the TSP. While the lack of proper dependency tracking often causes confusion and could result in inaccurate project status, TSP earned value tracking allows tasks to be worked in any order. As a result, team members are free to work on other issues (clearly identified in their plan) while they are awaiting the completion of a dependent task. The danger is that team members can also choose to complete all other tasks assigned to them first and leave the dependency task for last. This, in effect, creates dead time for other team members who depend on the incomplete task. The team solved this problem by closely coordinating with each other during the weekly team meetings [8].

The 309 SMXG project found that it was possible to run a TSP project within a UML environment.

Lessons Learned

After completion of the design, the lessons learned from implementation of the TSP include the following:

- Although the adjusted SLOC estimation worked for first pass through the project, it has since been found that it does not always correlate well to effort. The reason for this is that the UML tool does not consistently convert code in the *one-third user code* ratio presented above. The UML tool is highly dependent on the *whims* of the auto-generator when converting UML to SLOC. In future TSP iterations, the team has determined to find a new method of code counting and estimating that more accurately reflects the effort and time spent on UML. To find this new method, detailed statistical data is being gathered that reflects UML design objects and effort taken to produce them. In the meantime, various size metrics are being determined and gathered for code size proxy [8].
- The team determined that TSP does work in a UML auto-generated code environment, as long as size estimation issues are properly dealt with and the percent of time spent in each phase (design, code, test, etc.) is adjusted to increase the amount of time needed to design.
- When designing using UML and auto-generated code, part of the design may also be considered *implementation* or *coding*. As a result, the typical PSP phases must be modified to reflect that the design phase will now share tasking from the coding phase. On our project, this resulted in a separation of the code phase into two parts: design/code and code. The design/code reflected the auto-generated part of the UML, and the code reflected the user-entered portion of the UML implementation.

"The TSP practice of tracking progress at the personal level and then rolling that data up to the team level allowed this project to find and correct potential estimate issues before the project missed any schedule deadlines."

Conclusion

The TSP processes were very effective for this team. Not only did the introduction of the TSP bring the team's CMMI compliance quickly to Level 3 and beyond, but the structure and format of the processes allowed for better understanding of each team members' responsibilities and tasks involved in completion of the design project.◆

References

1. Webb, David R., and Watts Humphrey. "Using the TSP on the TaskView Project." CROSSTALK Feb. 1999 <www.stsc.hill.af.mil/crosstalk/1999/02/index.html>.
2. McHale, James, and Daniel S. Wall. "Mapping TSP to CMMI." Pittsburgh, PA: Software Engineering Institute, 22 June 2005.
3. Booch, Grady, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 2001.
4. Sanderfer, Lynn. "How and Why to Use the Unified Modeling Language." CROSSTALK June 2005 <www.stsc.hill.af.mil/crosstalk/2005/06/0506sanderfer.html>.
5. Humphrey, Watts S. A Discipline for Software Engineering. Addison-Wesley, 1995.
6. Tuma, David, and David R. Webb. "Personal Earned Value: Why Projects Using the Team Software Process Consistently Meet Schedule Commitments." CROSSTALK Mar. 2005 <www.stsc.hill.af.mil/crosstalk/2005/03/0503tuma.html>.
7. Webb, David R. "All the Right Behavior." CROSSTALK Sept. 2002 <www.stsc.hill.af.mil/crosstalk/2002/09/webb.html>.
8. Webb, David R. "Managing Risk With TSP." CROSSTALK June 2000 <www.stsc.hill.af.mil/crosstalk/2000/06/webb.html>.

About the Authors

David R. Webb is a senior technical program manager for the Software Division of Hill Air Force Base in Utah, a Capability Maturity Model® Level 5 organization. He is a project management and process improvement specialist with more than 18 years of technical, program management, and process improvement experience with Air Force software. Webb is a Software Engineering Institute-authorized instructor of the Personal Software Process℠, a Team Software Process℠ launch coach, and has worked as an Air Force section chief, systems software engineer, and test engineer. Webb has a bachelor's degree in electrical and computer engineering from Brigham Young University.

7278 4th ST
BLDG 100 RM 109
Hill AFB, UT 84056
Phone: (801) 940-7005
DSN: 940-7005
E-mail: david.webb@hill.af.mil

Ilya Lipkin is an electronics engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Center, Hill Air Force Base, Utah. His current research interests include artificial intelligence, human knowledge capture and analysis, neural networks, fuzzy logic, user interface design, software engineering, and customer relations management. Lipkin has a Bachelor of Science in computer engineering from the University of Toledo, a Master of Science in computer engineering from the University of Michigan, and is a doctoral candidate at the University of Toledo Business School.

7278 4th ST
BLDG 100 RM 109
Hill AFB, UT 84056
Phone: (801) 586-4477
Fax: (801) 586-2042
E-mail: ilya.lipkin@hill.af.mil

Evgeniy Samurin-Shraer is an electrical engineer at the 309th Software Maintenance Group at the Ogden Air Logistic Center, Hill Air Force Base, Utah. His current research interests include antenna design, resonance frequency circuits, and application of the microwave theory to the biomedical problems. Samurin-Shraer has a Bachelor of Science in electrical engineering and a Master of Science in electrical engineering from the University of Toledo.

7278 4th ST
BLDG 100 RM 109
Hill AFB, UT 84056
Phone: (801) 586-2048
Fax: (801) 586-2042
E-mail: evgeniy.samurin-shraer@hill.af.mil

Letter to the Editor

Dear CrossTalk Editor,
Kevin Stamey opened the November 2005 issue with these remarks in his "From the Sponsor" column:

> ... other engineering design disciplines have been in place for centuries; however, software engineering is still relatively new. The discipline of software design has only been matured for a few decades. It wasn't until the 1960s that the first software products hit the marketplace ... Our dominant programming language, C++, didn't emerge until the 1980s ...

The relative newness of software engineering is often cited when explaining the frustrations of the ongoing software crisis. However, the fact that current practices have only been around for a few decades, is that really extraordinary? Is our phenomenal growth all that unique? And have all the other engineering disciplines really been around for centuries?

Aeronautical and aerospace engineering may not be as new as software engineering, but there are certainly not centuries of experience in those fields. Not long ago, most aircraft were propeller-driven, and we referred to the sound barrier. Electrical engineering cannot be considered a centuries-old discipline unless you start with Ben Franklin's kite-and-key experiments.

Many of software engineering's principal tools have indeed been in place for a relatively short time, but isn't that true of most engineering disciplines? Niels Bohr's simplistic model of the atom is less than 100 years old. Physicists are continually discovering new particles; researchers are only beginning to explore the possibilities of quantum computing. Huge advances have been made by materials scientists, meaning circuitry and silicon technologies have undergone several significant advances in a relatively short time.

Indeed, our newness presents some formidable challenges, and provides fodder for intense debate. But we ought to avoid emphasizing that this newness makes us unique, or that our needing to adapt to rapidly evolving technologies and standards is somehow exclusive. Such naiveté presents us as making excuses for our shortcomings rather than boldly confronting challenges.

The first transistor was fabricated in the 1940s, and the first rudimentary integrated circuits were fabricated in the 1950s, about the same time that early compilers came into being. Software engineers don't need more time for their field to mature; like others in technological and engineering fields, we are challenged to advance and progress in a disciplined yet rapid fashion to keep up with the monumental advances occurring in the world around us.

There are several aspects of software engineering that set us apart from other engineering disciplines. Most notably, our end product is tied to the virtual world, not the physical world. As such, our discipline is governed less by the laws of physics, and we don't rely on equations as fundamental, foundational truths. This makes it harder to build upon the previous work of theoreticians in a predictable way – something that I think better explains our slow maturation than our relative newness.

John Reisner
Air Force Institute of Technology
<jreisner@gimail.af.mil>

Maturing the PSP: Developing a Body of Knowledge and Professional Certification for PSP-Trained Software Developers

Dr. Marsha Pomeroy-Huff
Software Engineering Institute

In the decade since its introduction, the Personal Software Process℠ (PSP℠) methodology has been adopted by thousands of individuals and dozens of major corporations the world over. As adoption of the PSP continues to grow, it has become critical to delineate and document the core skills and knowledge that set PSP practitioners apart from other software engineers. The PSP Body of Knowledge Vers. 1.0 was released by the Software Engineering Institute in September 2005 to serve this purpose and to provide individuals, organizations, and academic institutions with an objectively defined set of guidelines against which they can measure individual capabilities of engineers as well as assess or develop content for PSP-related training courses, curricula, or credential programs.

In his book "Software Craftsmanship: The New Imperative" [1], Pete McBreen makes the case that the software engineering profession is based on the wrong metaphor: Instead of an industrial model that views programmers as interchangeable parts in an assembly line development strategy that uses a one-size-fits-all process, software development should be classified as a form of craftsmanship. McBreen says that better software projects and higher-quality products will result only when software development shifts from applications of industrial solutions to software problems – for example, shifting from throwing volumes of personnel at troubled projects in an effort to get the work done faster – to focusing attention on practices typically associated with fine crafting. This includes rewarding developers for the disciplined implementation and mastery of their profession; allowing professionals to use situationally appropriate, flexible processes; encouraging people to work together as members of small, collaborative teams; and encouraging people to stop mass-producing good-enough software and to start creating high-quality work to which they would be proud to sign their names.

These tenets are consistent with the philosophy underlying the Software Engineering Institute's (SEI℠) Personal Software Process℠ (PSP℠), a proven effective method that takes just such a craftsman-like approach to software engineering. PSP-trained developers strive for quality at the individual level, take pride in and responsibility for the work that they produce, and use defined and disciplined tailorable processes to produce high-quality work on planned schedules for predictable costs. The PSP also provides flexibility since it is language-independent, allows for tailoring to mesh with established organizational processes, and can be implemented at any (or every) phase of the software development life cycle. It can also be tailored to fit various project needs and scaled up or down to address the needs of teams ranging in size from three or four individuals to larger teams of teams.

> *"... a body of knowledge is defined as a document generated by masters of a particular profession to identify and delineate the concepts, facts, and essential skills that professionals and practitioners in that profession are expected to have mastered."*

Since the introduction of the PSP methodology to the developer community in 1993, its effective use in a variety of academic and industrial settings has been documented in numerous peer-reviewed journal articles and technical reports [2, 3]. This has led to exponential growth in the adoption of PSP during the last decade (along with growing adoption of its sister technology, the Team Software Process℠ [TSP℠]) by many leading organizations in the software development community.

The PSP Body of Knowledge

During the past decade, a variety of articles, books, and conference proceedings have documented the successful implementation of PSP methodologies in a variety of settings. The PSP technology has now reached a level of maturity sufficient to allow – even require – further refinements to be made by the community of PSP users, academic institutions, and certification entities. To encourage and facilitate this effort, the SEI's Software Engineering Process Management program authorized the creation of a PSP Body of Knowledge (PSP BOK).

For the purposes of this article, a *body of knowledge* is defined as a document generated by masters of a particular profession to identify and delineate the concepts, facts, and essential skills that professionals and practitioners in that profession are expected to have mastered. The PSP BOK is derived from published literature and documented reports of practitioner experience. It is meant to be a concise, one-stop reference source that provides an overview of the knowledge and skill areas that are considered by expert PSP users to reflect the best practices and essential core abilities required for successful implementation of the PSP methodology. The PSP BOK is not meant to be an exhaustive list of every supporting detail, fact, or formula used in the PSP, nor should it be regarded as a *Cliffs Notes*-type replacement for the original source documents describing the PSP methodology [4, 5, 6]. The main purposes of the PSP BOK are the

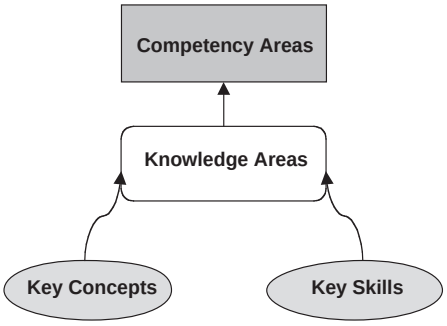


Figure 1: *BOK Architectural Hierarchy*

following:

- Define and characterize the basic essential competencies and standard practices that a PSP-trained professional is expected to master.
- Provide a consistent view of PSP in the community.
- Delineate the skills and knowledge that set PSP-trained professionals apart from other software developers.
- Encourage and facilitate the establishment of PSP-based courses and/or software engineering programs in academic institutions at both the undergraduate and graduate levels.
- Establish an objective baseline for developing and/or assessing PSP courses and curricula.
- Provide an established baseline for PSP certification or credential programs.
- Enable employers to objectively assess the software skills and capabilities of their software developers and project development teams.

The PSP BOK is organized according to an architectural hierarchy, in which related key concepts and skills are grouped into knowledge areas that, in turn, are grouped into competency areas (see Figure 1). The PSP BOK uses the term *key concept* to describe the intellectual aspects of the PSP content; that is, the essential information, facts, terminology, and philosophical components of a technology. The term *key skill* refers to the ability of an engineer to apply knowledge to perform a task. Together, a group of related key concepts and key skills form a *knowledge area* and, in turn, groups of related knowledge areas constitute a *competency area*.

Version 1.0 of the PSP BOK is composed of seven competency areas:

1. Foundational knowledge from other disciplines (e.g., statistics).
2. Basic concepts of the PSP.
3. Size measuring and estimating.
4. Making and tracking project plans.
5. Planning and tracking software quality.
6. Software design.
7. PSP process extensions.

Each competency area is composed of between three and seven knowledge areas which, in turn, contain one or more component key concepts, key skills, or a combination of the two, along with a brief description of the relevance and/or application of the concept or skill. The PSP BOK extracts in Tables 1, 2, and 3 provide an example of the structure and content of the BOK. The complete document can be downloaded from the SEI's publications Web site, <www.sei.cmu.edu/publications/documents/05.reports/05sr003.html>.

As PSP use continues to grow, further additions and evolutions to the BOK will inevitably follow, particularly in the competency area covering process extensions. The PSP BOK authors invite knowledgeable PSP users to submit suggestions and input for future revisions to the BOK.

Table 1: *Example of BOK Competency Area Content and Organization*

Competency Area Number and Name	7. Process Extensions and Customization
Competency Area Description	This competency area describes the modifications to the PSP that are required when scaling up from smaller programs to larger ones, when working with unfamiliar situations or environments, or when moving to team-based development instead of working alone.
Knowledge Areas	7.1 Defining a Customized Personal Process 7.2 Process Evolution 7.2 Advanced Process Applications
References	[Horn 90] [Humphrey 95, pp. 483-485, 725-740] [Humphrey 05a, Chapter 13]

Competency Area 7: Process Extensions and Customization

Table 2: *Example of BOK Knowledge Area Content and Organization*

Knowledge Area Number and Name	Description
7.1 Defining a Customized Personal Process	A defined process should not be regarded as *one size fits all*. This knowledge area addresses situations in which processes must be customized to meet changes in needed outputs or developed from the ground up to address new situations or environments.
7.2 Process Evolution	A process cannot be evolved to fit changing needs or situations until the current process accurately represents what is actually done when using that process. This knowledge area addresses the activities involved with incrementally evolving an initial process into one that is an accurate and complete description of the actual process.
7.3 Advanced Process Applications	Experienced PSP users may encounter situations when the original PSP processes may not be conveniently applicable for planning and developing a product. Modifications of the PSP, such as the Prototype Experimental Process (PEP) and the Product Maintenance Process (PMP), allow the application of PSP concepts and skills to such situations.

Description of the Process Extensions and Customization Knowledge Areas

Applications of the PSP BOK

The PSP BOK is intended for use in a variety of professional, industrial, and academic settings. For example, it can be used as a basis for credentialing practitioners who have attained proficiency in all of the key concepts and skills that the BOK comprises. This section discusses several potential uses of the PSP BOK in more detail.

Information contained in the PSP BOK can be used by individuals and employers in assessing the skills of software development professionals. Since personnel costs constitute well over half the budget of most software development projects, the skills of software developers have a large effect on the cost, schedule, and quality of the products produced. Employers are increasingly interested in hiring individuals who possess skills that allow them to work in a variety of software domains using disciplined and replicable methods – such as PSP – to consistently turn out well-crafted and high-quality work.

The essential concepts and skills required for proficient implementation of the PSP methodology are delineated in the PSP BOK. Therefore, the docu-

ment can be used to assist software engineering professionals in assessing their own skills and proficiencies and in identifying areas in which they may need further improvement. The PSP BOK can also be used by employers who want to establish an objective baseline for measuring the software development skills and capabilities of their engineers and product development teams. By understanding software engineering best practices, the industry can implement improvement efforts within its organizations, thereby achieving higher quality products and better management of costs and schedules.

The PSP BOK can assist academic institutions in updating software engineering or computer science curricula to reflect current software development practices used in industry. With the growing adoption of PSP and TSP, it is likely that employers will begin to require that newly hired developers possess PSP skills. The PSP BOK provides academic institutions with guidelines that will help them prepare students to work in industries that require individuals who are able to follow disciplined software development practices. Some institutions may choose to offer a PSP course, while others may choose to integrate PSP into several of their courses.

In both cases, institutions can use the guidance provided by the PSP BOK to ensure that students receive adequate instruction in and experience with fundamental PSP concepts and practices. PSP instruction offered by academic institutions will also provide a benchmark for industrial or commercial entities that may be interested in developing training programs based on the PSP BOK. Academic instruction in the BOK competencies, knowledge areas, key concepts, and key skill areas also provides a baseline for assessing the quality of instruction offered through industrial or commercial training or other such venues.

The PSP BOK may also serve as a foundation for creation of credentials or certifications that serve as a hallmark of a professional's ability to craft high-quality products. Certification is one of the most widely used mechanisms employed by a profession to make explicit the core set of knowledge and skills that a professional is expected to master. Certification also establishes a mechanism for objectively assessing mastery of those core competencies, and provides a foundation for continuing qualification of individual professionals.

Key Concept Number and Name	Description
7.3.1 Prototype Experimental Process (PEP)	Use the PEP when working in unfamiliar programming environments or when building prototype systems with loosely defined or poorly understood requirements.
7.3.2 Product Maintenance Process (PMP)	Use the PMP when making modifications, enhancements, or repairs to legacy systems with large or defective base code.

Knowledge Area 7.3: Advanced Process Applications
Experienced PSP users may encounter situations when the original PSP processes may not be conveniently applicable for planning and developing a product. Modifications of the PSP such as the Prototype Experimental Process (PEP) and the Product Maintenance Process (PMP) allow the application of PSP concepts and skills to such situations.

Table 3: *Example of BOK Key Concepts Content and Organization*

Thus, the PSP BOK provides an objective and concise description of the necessary skills and knowledge needed for attaining a *craftsman* level of competence in software development. The successful completion of courses, curricula, or credential programs that are based on the content of the PSP BOK provides a tangible measure of an individual's proficiency as part of an elite guild of software development crafters.

Conclusion

As PSP has gained acceptance by a broad spectrum of users within the software engineering community, the methodology has achieved sufficient stability and maturity to necessitate the documentation of the core skills and knowledge that set PSP practitioners apart from other software engineers. The PSP BOK Vers. 1.0 was released by the SEI in September 2005 to serve this purpose and to provide individuals, organizations, and academic institutions with an objectively defined set of guidelines against which they can measure individual capabilities of engineers, as well as determining the content required for effective PSP-related training courses or curricula. Completion of courses, curricula, or credential programs based on the PSP BOK will allow software professionals to effectively demonstrate the specialized knowledge and skills that set them apart from other programmers and allow them to consistently produce high-quality, well-crafted products.◆

References

1. McBreen, Pete. Software Craftsmanship: The New Imperative. Addison-Wesley, 2001.
2. Hayes, Will, and James Over. The Personal Software Process: An Empirical Study of the Impacts of PSP on Individual Engineers. Pittsburgh, PA: Carnegie Mellon University, Dec. 1997.
3. Kamatar, Jagadish, and Will Hayes. "An Experience Report on the Personal Software Process." IEEE Software 17.6 (Nov./Dec. 2000): 85-89.
4. Humphrey, Watts. A Discipline for Software Engineering. Addison-Wesley, 1995.
5. Humphrey, Watts. Introduction to the Personal Software Process. Addison-Wesley, 1997.
6. Humphrey, Watts. PSP: A Self-Improvement Process for Software Engineers. Addison-Wesley, 2005.

About the Author

Marsha Pomeroy-Huff, Ed.D., is a member of the technical staff at the Software Engineering Institute (SEI℠). Since joining the SEI in 1992, she has focused her work in the area of technology transition, specializing in development of educational products for software engineering practitioners. She is currently a member of the Personnel Software Process℠ (PSP℠)/ Team Software Process℠ (TSP℠) Initiative and the PSP Professional Certification team, and is primary instructor for SEI's Introduction to Personal Process course. Pomeroy-Huff has a doctorate in instructional design and technology from the University of Pittsburgh.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
Phone: (412) 268-3423
Fax: (412) 268-5758
E-mail: mph@sei.cmu.edu

Understanding the Logic of System Testing

Dr. Yuri Chernak
Valley Forge Consulting, Inc.

What do system testing and mathematics have in common? They both deal with proofs. This article discusses the logic of system testing, and the steps to construct valid proofs that testers need to perform for their conclusions about the quality of a software product.

Test case is a fundamental term used by all software testers. Numerous published sources, including the Institute of Electrical and Electronics Engineers (IEEE) Standard 610 (IEEE Std. 610), define what a test case is, techniques to design test cases, and templates to document them. However, testers in the field still find these definitions confusing, and they frequently mean different things when referring to test cases.

A common misunderstanding of test cases can be a symptom of a larger issue – a misunderstanding of the logic of software testing. The main purpose of software testing can be defined as exploring the software product to derive and report valid conclusions about its quality and suitability for use. In this regard, software testing is similar to mathematics – they both need proofs for their conclusions. However, mathematicians surpass software testers in deriving and proving their conclusions thanks to their skill in using a powerful tool called *deductive reasoning*. To construct valid arguments, logicians have developed proof strategies and techniques based on the concepts of symbolic logic and proof theory [1, 2, 3].

On critical software projects, testers have always been required to present valid evidence supporting their conclusions about the product's quality. The recent Sarbanes-Oxley Act[1] makes this requirement much more important going forward. In this article, I discuss the logic of one of the conventional levels of testing – system test [4] – and propose a formal approach to constructing valid arguments supporting testers' conclusions. Finally, understanding the system test logic can help testers better understand the meaning of test cases.

Proofs and Software Testing

Software testers have always dealt with proofs on their projects. One example can be concluding that a system passed testing. As testers can never prove the absence of bugs in a software product, their claim that a system passed testing is conditional upon the evidence and arguments supporting such a claim. On critical projects, either the project's manager, end-users, or a compliance department commonly require documented test cases and test execution logs to be used as grounds for supporting testers' conclusion that a software product passed testing.

Another example is reporting a system failure. Regardless whether it is formal testing or unscripted exploratory testing, testers are required to document and report the defects they find. By reporting a defect, a tester first claims that a certain system feature failed testing and, second, presents an argument in the form of a defect description to support the claim. Such an argument should be logically valid to be sufficiently convincing for developers.

Deriving conclusions and presenting valid proofs, also known in mathematics as logical arguments, is frequently not a trivial matter. That is why mathematicians use *deductive reasoning* as a foundation for their strategies and techniques to derive conclusions and present valid arguments. Deductive reasoning is the type of reasoning used when deriving a conclusion from other sentences that are accepted as true [3]. As I discuss in this article, software testers can also benefit from using deductive reasoning. First, they can better understand the logic of software testing and, second, they can construct valid proofs better supporting their conclusions about product quality.

Applying Deductive Reasoning to Software Testing

In mathematics, the process of deriving a conclusion results in presenting a deductive argument or proof that is defined as a convincing argument that starts from the premises and logically deduces the desired conclusion. The proof theory discusses various logical patterns for deriving conclusions, called *rules of inference*, that are used as a basis for constructing valid arguments [2, 3]. An argument is said to be valid if the conclusion necessarily follows from its premise. Hence, an argument consists of two parts – a conclusion, and premises offered in its support. Premises, in turn, are composed of an implication and evidence. Implications are usually presented as conditional propositions, for example, (*if A, then B*). They serve as a bridge between the conclusion and the evidence from which the conclusion is derived [1]. Thus, the implication is very important for constructing a logical argument as it sets the argument's structure and meaning. In the following, I will apply this concept to software testing and identify the argument components that can be used in testing to construct valid proofs.

In software testing, we derive and report conclusions about the quality of a product under test. In particular, in system testing a common unit of a tester's work is testing a *software feature* (also known in Rational Unified Process as *test requirement*); the objective is to derive a conclusion about the feature's testing status. Hence, the feature status, commonly captured as *pass* or *fail*, can be considered a conclusion of the logical argument. To derive such a conclusion, test cases are designed and executed. By executing test cases, information is gained, i.e., evidence is acquired that will support the conclusion. To derive a valid conclusion, also needed are implications that in system testing are known as a feature's pass/fail criteria. Finally, both the feature's pass/fail criteria and the test case execution results are the premises from which a tester derives a conclusion. The lack of understanding of how system testing logic works can lead to various issues – some of the most common of which I will discuss next.

Common Issues With Testing Logic

Issue 1: Disagreeing About the Meaning of Test Cases

Software testers frequently disagree about the meaning of *test cases*. Many testers would define a test case as the whole set

of information designed for testing the same software feature and presented as a test-case specification. Their argument is that all test inputs and expected results are designed for the same objective, i.e., testing the same feature, and they all are used as supporting evidence that the feature passed testing.

For other testers, a *test case* consists of each pair – input and its expected result – in the same test-case specification. In their view, such a test-case specification presents a set of test cases. To support their point, they refer to various textbooks on test design, for example [4, 5], that teach how to design test cases for boundary conditions, valid and invalid domains, and so on. Commonly, these textbooks focus their discussion on designing test cases that can be effective in finding bugs. Therefore, they call each pair of test input and its expected output a test case because, assuming a bug is in the code, such a pair provides sufficient information to find the bug and conclude that the feature failed testing.

Despite these different views, both groups actually imply the same meaning of the term *test case*: information that provides grounds for deriving a conclusion about the feature's testing status. However, there is an important difference: The first group calls *test case* the information that supports the feature's pass status, while the second group calls *test case* the information that supports the feature's fail status. Such confusion apparently stems from the fact that all known definitions of the term test case do not relate it to a feature's pass/fail criteria. However, as the discussion in the sidebar shows, these criteria are key to understanding the meaning of test cases.

Issue 2: Presenting an Argument Without a Conclusion

This issue is also very common. As discussed earlier, an important part of a logical argument is its conclusion. However, a lack of understanding of this concept can lead to presenting arguments without conclusions. On a number of projects, I have seen testers produce test case documentation in the form of huge tables or Excel spreadsheets listing their test cases. In such tables, each row shows a test case represented by a few columns such as test case number, test input, expected result, and test case execution (pass/fail) status. What is missing in this documentation is a description of what features testers intend to evaluate using these test cases. As a result, it is difficult to judge the validity and verify the completeness of such test cases as the underlying purpose for which they were designed is not known. Such documentation suggests that the testers who designed it do not completely understand the logic of software testing.

Issue 3: Presenting an Argument Without an Implication

This issue also stems from a lack of understanding of the structure of a logical argument, specifically that having an implication is necessary for deriving a valid conclusion. In software testing, such implications are a feature's pass/fail criteria. The issue arises when such criteria are either forgotten or not clearly defined and understood by testers. This can lead to a situation where testers lose sight of what kind of conclusions they need to report. As a result, instead of deriving a conclusion about the feature and then reporting its testing status, they report the status of each executed test case. This situation presents an issue illustrated in the following example.

Let us assume a tester needs to test 10 software features, and he or she designed 10 test cases for each of the features under test. Thus, the entire testing requires executing 100 test cases. Now, while executing test cases, the tester found that one test case failed for each of the features. In our example, the tester did not define and did not think about the feature pass/fail criteria. Instead, the tester reported to a project manager the testing status for each executed test case. Thus, at the end of the testing cycle, the results show that 90 percent of testing was successful. When seeing such results, a manager would be fairly satisfied and could even make a decision about releasing the system.

The project manager would see a completely different picture if the features' pass/fail criteria were not forgotten. In this case, the testers would report the testing status for each feature as

What Do We Call a Test Case?

Most of the published sources defining the term *test case* follow the definitions given in the Institute of Electrical and Electronics Engineers (IEEE) Standard 610 (IEEE Std. 610):

a) **Test Case:** A set of test inputs, execution conditions, and expected results developed for a particular objective such as to exercise a particular program path or to verify compliance with a specific requirement.
b) **Test Case:** Documentation specifying inputs, predicted results, and a set of execution conditions for a test item.

Despite the fact that this standard was published many years ago, testers in the field still do not have a consistent understanding of the meaning of test cases. To better understand this meaning, we can use the concept of deductive reasoning. Following this concept, system testing can be viewed as a process of deducing valid conclusions about the testing status of system features based on evidence acquired by executing test cases. Hence, the main purpose of executing test cases is to gain information about the system implementation. This information can be used together with the feature's pass/fail criteria to derive and support conclusions about the status of feature testing. The feature's pass/fail criteria are important implications in the testing argument that determine the meaning of testers' conclusions. These criteria are a link between a tester's conclusion about the feature status and the test cases used to support the conclusion. *Hence, the meaning of test cases follows from the definition of the feature's pass/fail criteria.*

In system testing, the mission is finding and reporting software defects; the feature fail criterion is commonly defined as, "If any of the feature's test cases fails, then the feature fails testing." What follows from this implication is that a test case is information that is sufficient to identify a software defect by causing a system feature to fail. The feature's pass criterion can further explain the meaning of test cases. It is commonly defined as, "The feature passes the test only if all of its test cases pass testing." According to this definition, we imply that the system feature passed testing only if the whole group of its test cases passed testing. Hence, test cases are used as collective evidence to support the feature's pass status. It should be noted, however, that this interpretation of the test-case meaning refers to the system test only. In contrast, in acceptance testing a testing mission and pass/fail criteria can be defined differently from the system testing. Correspondingly, the meaning of test cases can be different as well.

If the IEEE definitions of test cases are examined again, we can see that these definitions are not specific to a particular testing mission, nor are they explicit about which testing conclusion, i.e., pass or fail, a test case is intended to support. Instead, they focus primarily on the test case structure: test inputs and expected results. As a result, these definitions alone and without the feature's pass/fail criteria lack clarity about the test case purpose and meaning.

opposed to each test case. If the feature fail criterion were defined as, "If any of the feature's test cases fails, then the feature fails testing," then the testing end-result in our example would have been quite the opposite and have shown that none of the software features passed testing; they all should be re-tested when the bugs are fixed.

An Approach to Constructing A Valid Proof

Constructing a valid proof in system testing can be defined as a four-step procedure. The following sections discuss each step in detail, and explain how to construct a valid argument to support a testing conclusion.

Step 1: Define a Conclusion of the Argument

In constructing a proof, always begin by defining what needs to be proven, i.e., the conclusion. In system testing, the ultimate goal is to evaluate a software product. This is achieved by decomposing the entire functional domain into a set of functional features where each feature to be tested is a unit of a tester's work that results in one of the two possible conclusions about a feature's testing status *pass* or *fail*. At any given time, only one of the two conclusions is valid.

The term *software feature* is defined in the IEEE Std. 610 as follows:

a) A distinguished characteristic of a software item.
b) A software characteristic specified or implemented by requirements documentation.

From a tester's perspective, a software feature means any characteristic of a software product that the tester believes might not work as expected and, therefore, should be tested. Deciding what features should be in the scope of testing is done at the test-planning phase and documented in a test plan document. Later, at the test design phase, the list of features is refined and enhanced based on a better understanding of the product's functionality and its quality risks. At this phase, each feature and its testing logic are described in more detail. This information is presented either in test design specifications and/or in test case specifications.

The test design specification commonly covers a set of related features; whereas, the test case specification commonly addresses testing of a single feature. At this point, a tester should already know which quality risks to focus on in feature testing. Understanding the feature's quality risks, i.e., how the feature can fail, is important for designing effective test cases that a tester executes to evaluate the feature's implementation and derive a conclusion about its testing status. Performing Step 1 can help testers avoid Issue No. 2 as discussed earlier.

Step 2: Define an Implication of the Argument

The next important step is to define an implication of an argument. An implication of a logical argument defines an important relation between the conclusion and the premises given in its support. Correspondingly, the feature's pass/fail criteria define the relation between the results of test-case execution and the conclusion about the feature's evaluation status.

According to the IEEE Std. 829, the feature's pass/fail criteria should be defined in the test design specification; this standard provides an example of such a specification. However, it does not provide any guidance on how to define these criteria, apparently assuming this being an obvious matter that testers know how to handle. Neither do the textbooks on software testing methodology and test design techniques. Contrary to this view, I feel that defining these criteria is one of the critical steps in test design that deserves a special consideration. As I discussed earlier and illustrated as Issue No. 3, the lack of understanding of the role and meaning of the feature's pass/fail criteria can lead to logically invalid testing conclusions in system testing. Also, as discussed in the sidebar, from the well-defined implications, i.e., the features' pass/fail criteria, testers can better understand the meaning of test cases and avoid the confusion discussed earlier as Issue No. 1.

The rationale for defining the feature's pass/fail criteria stems from the system test mission that can be defined as *critically examining the software system under the full range of use to expose defects and to report conditions under which the system is not compliant with its requirements*. As such, the system test mission is driven by the assumption that a system is not yet stable and has bugs; the testers' job is to identify conditions where the system fails. Hence, our primary goal in system testing is to prove that a feature fails the test. To do that, testers develop ideas about how the feature can fail. Then, based on these ideas, testers design various test cases for a feature and execute them to expose defects in the feature implementation. If this happens, each test case failure provides sufficient grounds to conclude that the feature failed testing. Based on this logic, the feature's fail criterion can be defined as, "If any of the feature's test cases fail, then the feature fails testing." In logic, this is known as a sufficient condition (*if A, then B*). The validity of this implication can also be formally proved using the truth-table technique [1]; however, this goes beyond the scope of this article.

Defining the feature's pass criterion is a separate task. In system testing, testers can never prove that a system has no bugs, nor can they test the system forever. However, at some point and under certain conditions they have to make a claim that a feature passed testing. Hence, the supporting evidence, i.e., the test case execution results, can only be a necessary (*C, only if D*), but not a sufficient condition of the feature's pass status. Based on this logic, the feature's pass criterion can be defined as, "The feature passes the test only if all of its test cases pass testing." In this case, the feature's pass criterion means two things:

a) The feature pass conclusion is conditional upon the test execution results presented in its support.
b) Another condition may exist that could cause the feature to fail.

Step 3: Select a Technique to Derive a Conclusion

Once we have defined all components of a testing argument, the next step is to select a technique that can be used to derive a valid conclusion from the premises. The word *valid* is very important at this point as we are concerned with deducing the conclusion that logically follows from its premises. In the proof theory, such techniques are known as *rules of inference* [1, 2]. By using these rules, a valid argument can be constructed and its conclu-

Table 1: *Deriving a Feature Fail Conclusion*

Modus Ponens Form	Testing Argument Form
1. If A, then B – *means* ⟶	1. If any test case fails, then a feature fails (*implication*).
2. A is true – *means* ⟶	2. We know that at least one test case failed (*evidence*).
3. Then B is true – *means* ⟶	3. Then the feature fails the test (*conclusion*).

sion deduced through a sequence of statements where each of them is known to be true and valid. In system testing, there are two types of conclusions – a feature fail status and a feature pass status. Correspondingly, for each of these conclusions, a technique to construct a valid argument is discussed. On software projects, testers should discuss and define the logic of constructing valid proofs before they begin their test design. For example, they can present this logic in the *Test Approach* section of a test plan document.

Deriving a Feature Fail Conclusion

I defined the feature's fail criterion as a conditional proposition in the form (*if A, then B)*, which means if any of the test cases fail, then testers can conclude that the feature fails as well. This also means that each failed test case can provide sufficient evidence for the conclusion. In this case, a valid argument can be presented based on the rule of inference, known as *Modus Ponens* [1, 2]. This rule is defined as a sequence of three statements (see Table 1). The first two statements are premises known to be true and lead to the third statement, which is a valid conclusion.

Deriving a Feature Pass Conclusion

The feature's pass criterion was defined as a conditional proposition in the form (*C, only if D)*, which means a feature passes the test only if all of its test cases pass testing. This also means that such a conclusion is derived only when all of the feature's test cases have been executed. At this point, the feature status can be either pass or fail, but not anything else. Hence, the rule of inference can be used, known as *Disjunctive Syllogism* [1, 2], which is presented as three consecutive statements that comprise a valid argument (see Table 2).

Step 4: Present an Argument for a Conclusion

At this point, there is a clear plan on how to construct valid arguments in system testing. The actual process of deriving a testing conclusion begins with executing test cases. By executing test cases, the testers can learn the system's behavior and analyze the feature implementation by comparing it to its requirements captured by expected results of test cases. As a result, testers can acquire evidence from which they can derive and report a valid testing conclusion, i.e., a feature pass or fail testing status.

Concluding a Feature Fail Status

The feature fail criterion is defined as, "If any of the test cases fails, then the feature fails testing." According to the Modus Ponens rule, this means that each failed test case provides grounds for the valid conclusion that the feature has failed testing. As a feature can fail on more than one of its test cases, after finding the first defect a tester should continue feature testing and execute all of its test cases. After that, the tester should report all instances of the feature failure by submitting defect reports, where each defect report should be a valid argument that includes the evidence supporting the feature fail status.

Disjunctive Syllogism Form	Testing Argument Form
1. Either P or Q is true – *means* ⟶	1. After all test cases have been executed, a feature status can be either fail (P) or pass (Q) (*implication*).
2. P is not true – *means* ⟶	2. We know that the feature did not fail the test for all of its test cases (*evidence*).
3. Then Q is true – *means* ⟶	3. Then the feature passes the test (*conclusion*).

Table 2: *Deriving a Feature Pass Conclusion*

On the other hand, if a given test case passed testing, the Modus Ponens rule does not apply, and there are no grounds for any conclusion at this point, i.e., the feature has neither passed nor failed testing. Finally, only when all of the feature's test cases have been executed should it be decided whether there are grounds for the feature pass status as discussed in the next section.

Concluding a Feature Pass Status

Obviously, if the feature has already failed, the pass status cannot have grounds. However, if none of the test cases failed, then the Disjunctive Syllogism rule can be applied. According to this rule, the fact that none of the test cases failed provides grounds for a valid conclusion: the feature passed testing. To support this claim, evidence is provided – test case execution results. However, this conclusion should not be confused with the claim that the feature implementation has no bugs, which we know is impossible to prove. The conclusion means only that the feature did not fail on the executed test cases that were presented as evidence supporting the conclusion.◆

References

1. Copi, I., and C. Cohen. Introduction to Logic. 11th ed. Prentice-Hall, 2002.
2. Bloch, E. Proof and Fundamentals. Boston: Birkhauser, 2000.
3. Rodgers, N. Learning to Reason. An Introduction to Logic, Sets, and Relations. John Willey & Sons, 2000.
4. Meyers, G. The Art of Software Testing. John Wiley & Sons, 1979.
5. Kit, E. Software Testing In the Real World. Addison-Wesley, 1995.

Note

1. The Sarbanes-Oxley Act <http://news.findlaw.com/hdocs/docs/gwbush/sarbanesoxley072302.pdf>.

Acknowledgements

I am grateful to the CROSSTALK reviewers, to the distinguished professor Sergei Artemov at the graduate center of the City University of New York, and to Robin Goldsmith at GoPro Management for their feedback and comments that helped me improve this article.

About the Author

Yuri Chernak, Ph.D., is the president and principal consultant of Valley Forge Consulting, Inc. As a consultant, Chernak has worked for a number of major financial firms in New York helping senior management improve the software testing process. Currently, his research focuses on aspect-oriented requirements engineering, use-case-driven testing, and test process assessment and improvement. Chernak is a member of the Institute of Electrical and Electronics Engineers (IEEE) Computer Society. He has been a speaker at several international conferences, and has published papers on software testing in IEEE publications and other professional journals. Chernak has a doctorate in computer science.

Valley Forge Consulting, Inc.
233 Cambridge Oaks ST
Park Ridge, NJ 07656
Phone: (201) 307-4802
E-mail: ychernak@yahoo.com

Availability, Reliability, and Survivability: An Introduction and Some Contractual Implications

Dr. Jack Murphy
DeXIsive Inc.

Dr. Thomas Ward Morgan
CACI Federal

This article is directed toward information technology professionals that enter into contractual agreements requiring service-level agreements (SLAs) that specify availability, reliability, or survivability objectives. Its purpose is to show a relationship between cost, performance, and SLA levels established by the customer.

Information technology (IT) outsourcing arrangements frequently employ service-level agreements (SLAs) that use terms such as availability and reliability. The intent is that the buyer requests a specific system availability and reliability (e.g., 98 percent to 99.9 percent, and 85 percent to 90 percent, respectively). The service provider is typically rewarded for exceeding specified limits and/or punished for falling below these limits.

In recent years, another term, survivability, has become popular and is used to express yet another objective: the ability of a system to continue functioning after the failure of one of its components. This article examines these terms so buyer and seller can understand and use them in a contractual context and designers/operators can choose optimal approaches to satisfying the SLAs.

The northeastern U.S. power grid failure in August 2003 drew attention to the availability, reliability, and survivability of business-critical IT systems. Catastrophe can be the catalyst for new thinking about the survivability of IT systems.

From the buyer's perspective, an increase in availability, reliability, and survivability comes at a price: 100 percent is not possible, but 98 percent might be affordable and adequate while 99.99 percent might be unaffordable or excessive. From the service provider's perspective, under-engineering or inadequate operating practices can result in penalties for failing to meet SLAs.

Availability

What Is Availability?

Availability is influenced by the following:

- **Component Reliability.** A measure of the expected time between component failures. Component reliability is affected by electromechanical failures as well as component-level software failure.
- **System Design.** The manner in which components are interrelated to satisfy required functionality and reliability. Designers can enhance availability through judicious use of redundancy in the arrangement of system components.
- **Operational Practices.** Operational practices come into play after the system is designed and implemented with selected components. Interestingly, after a system is designed, components are selected and the system is implemented. The only factor that can improve or degrade availability is operational practices.

Informally, system availability is the expected ratio of uptime to total elapsed time. More precisely, availability is the ratio of uptime and the sum of uptime, scheduled downtime, and unscheduled downtime:

$$A_1 = \frac{\text{Uptime}}{\text{Uptime + Downtime}} \qquad (1)$$

The formula (1) is useful for measuring availability over a given period of time such as a calendar quarter, but not very useful for predicting availability or engineering a system to satisfy availability requirements. For this purpose, system designers frequently employ a model based on mean time between failure (MTBF) and mean time to repair (MTTR), usually expressed in units of hours:

$$A_2 = \frac{\text{MTBF}}{\text{MTBF + MTTR}} \qquad (2)$$

Formula (2) is analogous to formula (1), but is based on statistical measures instead of direct observation. Most vendors publish MTBF data. MTTR data can often be collected from historical data. Interestingly, MTTR is partially within the control of the system operator. For example, the system operator may establish a strategy for spares or provide more training to the support staff to reduce the MTTR. Because of these factors, component vendors typically do not publish MTTRs. Finally, it should be noted that A_2 does not explicitly account for scheduled downtime.

The most common approach to include scheduled maintenance time is to include it in the total time represented in the mathematical model's denominator, thus reducing expected availability commensurately. This provides an additional challenge for the designer, but like MTTR it is somewhat controllable through operational procedures. If the system can be designed for only infrequent preventive maintenance, then availability is enhanced.

In most operational environments, a system is allowed to operate normally, including unscheduled outages, for some fixed period of time, t, after which it is brought down for maintenance for some small fraction of that time, λt (see Figure 1).

If the scheduled maintenance is periodic and on a predictable schedule, then following t hours there is a scheduled outage of λt so that the fraction of time that the system is not in maintenance is:

$$t / (t + \lambda t) = 1 / (1 + \lambda) \qquad (3)$$

Since the denominator in A2 does not include the time in preventive maintenance, an adjustment to formula (1) is needed whenever preventive maintenance is part of the operational routine. To include this time, the denominator needs to be increased by a factor of $1+ \lambda$ to accurately reflect the smaller actual availability expected. Stated differently, the denominator in A_2 needs to be modified to accurately represent all time, including operational (MTBF), in repair (MTTR), or in maintenance λ (MTBF + MTTR). The revised availability model in cases of scheduled downtime is:

$$A_3 = \frac{\text{MTBF}}{(1 + \lambda)(\text{MTBF + MTTR})} \qquad (4)$$

This model, like the model A_2, assumes independence between the model variables. In reality there may be some relationship between the variables MTBF, MTTR, and λ.

Availability Engineering

A complex system is composed of many interrelated components; failure of one component may not impact availability if the system is designed to withstand such a

failure, while failure of another component may cause system downtime and hence degradation in availability. Consequently, system design can be used to achieve high availability despite unreliable components.

For example, if Entity1 has component availability 0.9 and Entity2 has component availability 0.6, then the system availability depends on how the entities are arranged in the system. As shown in Figure 2, the availability of System1 is dependent on the availability of both Entity1 and Entity2.

In Figure 3, the system is unavailable only when both components fail. To compute the overall availability of complex systems involving both serially dependent and parallel components, a simple recursive use of the formulas in Figures 2 and 3 can be employed.

Thus far, the engineer has two tools to achieve availability requirements specified in SLAs: a selection of reliable components and system design techniques. With these two tools, the system designer can achieve almost any desired availability, albeit at added cost and complexity.

The system designer must consider a third component of availability: operational practices. Reliability benchmarks have shown that 70 percent of reliability issues are operations induced versus the traditional electromechanical failures. In another study, more than 50 percent of Internet site outage events were a direct result of operational failures, and almost 60 percent of public-switched telephone network outages were caused by such failures [1]. Finally, Cisco asserts that 80 percent of non-availability occurs because of failures in operational practices [2]. One thing is clear: Only a change in operational practices can improve availability after the system components have been purchased and the system is implemented as designed.

In many cases, operational practices are within control of the service provider while product choice and system design are outside of its control. Conversely, a system designer often has little or no control over the operational practices. Consequently, if a project specifies a certain availability requirement, say 99.9 percent (3-nines), the system architect must design the system with more than 3-nines of availability to leave enough *head space* for operational errors.

To develop a model for overall availability, it is useful to consider failure rate instead of MTBF. Let α denote the failure rate due to component failure only. Then $\alpha = 1/\text{MTBF}$. Also, let τ denote the total failure rate, including component failure as well as failure due to operational errors. Then $1/\tau$ is the mean time between failure when both component failure and failures due to operational errors are considered (MTBF_{Tot}).

If β denotes the fraction of outages that are operations related, then $(1 - \beta)\,\tau$ is the fraction of outages that are due to component failure. Thus:

$$(1 - \beta)\,\tau = \alpha$$
$$\text{So } \tau = \alpha / (1 - \beta) \text{ and}$$
$$\text{MTBF}_{\text{Tot}} = (1 - \beta) / \alpha \qquad (5)$$

The revised model becomes:

$$A_4 = \frac{\text{MTBF}_{\text{Tot}}}{(1 + \lambda)(\text{MTBF}_{\text{Tot}} + \text{MTTR})} \qquad (6)$$

where,

$$\text{MTBF}_{\text{Tot}} = \frac{1 - \beta}{\alpha}$$

When an SLA specifies an availability of 99.9 percent, the buyer typically assumes the service provider considers all forms of outage, including component failure, scheduled maintenance outage, and outages due to operational error. So the buyer has in mind a model like that defined by A_4. But the designer typically has in mind a model like A_2 because the design engineer seldom has control over the maintenance outages or operationally induced outages, but does have control over product selection and system design. Thus, the buyer is frequently disappointed by insufficient availability and the service provider is frustrated because the SLAs are difficult or impossible to achieve at the contract price.

If the system design engineer is given insight into λ, the maintenance overhead factor, and β, then A_2 can be accurately determined so that A_4 is within the SLA. For example, if $A_4 = 99$ percent, it may be necessary for the design engineer to build a system with $A_2 = 99.999$ percent availability to leave sufficient room for maintenance outages and outages due to operational errors.

Given an overall availability requirement (A_4) and information about λ and β, the design availability A_2 can be computed from formula (7). Note that high maintenance ratios become a limiting factor in being able to engineer adequate availability.

$$A_2 = \frac{(1 + \lambda)A_4}{1 + \beta\,((1 + \lambda)\,A_4 - 1)} \qquad (7)$$

For 3-nines of overall availability it is necessary to engineer a system for over 6-nines of availability (less than 20 seconds/year downtime due to component failure) even if only 50 percent of outages are the result of operational errors when the maintenance overhead is 0.1 percent. Engineering systems for 6-nines of availability may have a dramatic impact on system cost and complexity. It may be better to develop operational practices that minimize repair time and scheduled maintenance time.

Reliability

What Is Reliability?

There is an important distinction between the notion of availability presented in the preceding section and reliability. Availability is the expected fraction of time that a system is operational. Reliability is the probability that a system will be available (i.e., will not fail) over some period of time, *t*. It does not measure or model downtime. Instead reliability only models the time until failure occurs without concern for the time to repair or return to service.

Reliability Engineering

To model reliability, it is necessary to know something about the failure stochastic process, that is, the probability of failure before time, *t*. The Poisson Process, based upon the exponential probability distribution, is usually a good model. For

Figure 1: *Preventative Maintenance Cycle*

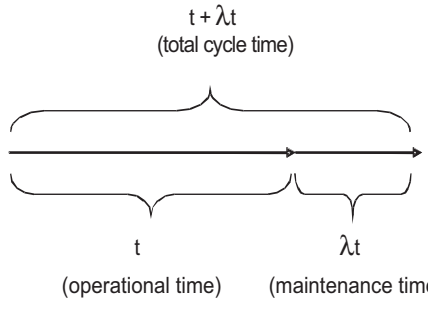


Figure 2: *Availability With Serial Components*

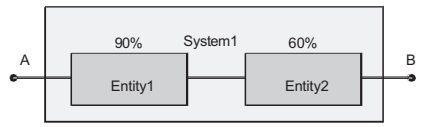
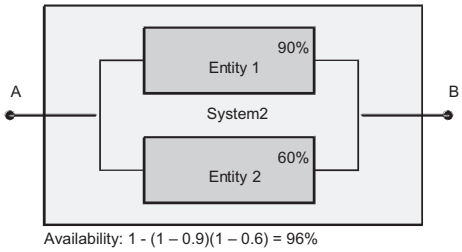


Figure 3: *Availability With Parallel Components*



this process, it is only necessary to estimate the mean of the exponential distribution to predict reliability over any given time interval. Figure 4 depicts the exponential distribution function and the related density function. As shown, the probability of a failure approaches 1 as the period of time increases.

If F(t) and f(t) are the exponential distribution and density functions respectively, then the reliability function R(t) = 1 – F(t). So that,

$$R(t) = 1 - \int_t^{\infty} f(t) = \int_0^t f(t) = \int_0^t \frac{1}{\theta} = e^{-t/\theta} \quad (8)$$

where,

θ = MTBF

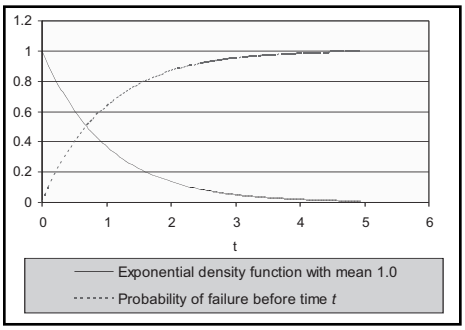
The mean of this distribution is θ, the MTBF. It can be measured directly from empirical observations over some historical period of observation or estimated using the availability models presented earlier.

Table 1 shows the reliability for various values of θ and *t*. Note that when t = MTBF, R(t) = 36.79 percent. That is, whatever the MTBF, the reliability over that same time period is always 36.79 percent.

Network components such as Ethernet switches typically have an MTBF of approximately 50,000 hours (about 70 months). Thus the annual reliability of a single component is about 85 percent (use t=12 and θ=70 in the formula above or interpolate using Table 2). If that component is a single point of failure from the perspective of an end workstation, then based on component failure alone the probability of outage for such workstations is at least 15 percent. When operational errors are considered, it is $MTBF_{Tot}$, not MTBF that determines reliability, so the probability of an unplanned outage within a year's time increases accordingly.

The preceding assumes that the exponential density function accurately models system behavior. For systems with periodic scheduled downtime, this assumption is invalid. At a discrete point in time, there is a certainty that such a system will be unavailable: $R(t) = e^{-t/\theta}$ for any $t < t_0$, where t_0 is the point in time of the next scheduled maintenance, and R(t) = 0 for $t \geq t_0$.

Figure 4: *Exponential Density and Distribution Functions With MTBF = 1.0*



Survivability

What Is Survivability?

Survivability of IT systems is a significant concern, particularly among critical infrastructure providers. Availability and reliability analysis assume that failures are somewhat random and the engineer's job is to design a system that is robust in the face of random failure. There is thus an implicit assumption that system failure is largely preventable.

Survivability analysis implicitly makes the conservative assumption that failure will occur and that the outcome of the failure could negatively impact a large segment of the subscribers to the IT infrastructure. Such failures could be the result of deliberate, malicious attacks against the infrastructure by an adversary, or they could be the result of natural phenomenon such as catastrophic weather events. Regardless of the cause, survivability analysis assumes that such events can and will occur and the impact to the IT infrastructure and those who depend on it will be significant.

Survivability has been defined as "the capability of a system to fulfill its mission in a timely manner, in the presence of attacks, failures, or accidents" [3]. Survivability analysis is influenced by several important principles:

- **Containment.** Systems should be designed to minimize mission impact by containing the failure geographically or logically.
- **Reconstitution.** System designers should consider the time, effort, and skills required to restore essential mission-critical IT infrastructure after a catastrophic event.
- **Diversity.** Systems that are based on multiple technologies, vendors, locations, or modes of operation could provide a degree of immunity to attacks, especially those targeted at only one aspect of the system.
- **Continuity.** It is the business of mission-critical functions that they must continue in the event of a catastrophic event, not any specific aspect of the IT infrastructure.

If critical functions are composed of both IT infrastructure (network) and function-specific technology components (servers), then both must be designed to be survivable. An enterprise IT infrastructure can be designed to be survivable, but unless the function-specific technologies are also survivable, irrecoverable failure could result.

Measuring Survivability

From the designers' and the buyers' perspectives, comparing various designs based upon their survivability is critical for making cost and benefit tradeoffs. Next we discuss several types of analysis that can be performed on a network design that can provide a more quantitative assessment of survivability.

Residual measures for an IT infrastructure are the same measures used to describe the infrastructure before a catastrophic event but are applied to the expected state of the infrastructure after the effects of the event are taken into consideration. Here we discuss four residual measures that are usually important:

- **Residual Single Points of Failure.** In comparing two candidate infrastructure designs, the design with fewer single points of failure is generally considered more robust than the alternative. When examining the survivability of an infrastructure with respect to a particular catastrophic event, the infrastructure with the fewer residual single points of failure is intuitively more survivable. This measure is a simple count.
- **Residual Availability.** The same availability analysis done on an undamaged infrastructure can be applied to an infrastructure after it has been damaged by a catastrophic event. Generally, the higher the residual availability of an infrastructure the more survivable it is with respect to the event being analyzed.
- **Residual Performance.** A residual infrastructure that has no single point of failure and has high residual availability may not be usable from the per-

Table 1: *Reliability Over t Months For MTBF Ranging From 3 to 96 Months*

R(t)		t (months)							
		1	2	4	8	12	24	48	96
MTBF (months)	3	71.65%	51.34%	26.36%	6.95%	1.83%	0.03%	0.00%	0.00%
	6	84.65%	71.65%	51.34%	26.36%	13.53%	1.83%	0.03%	0.00%
	12	92.00%	84.65%	71.65%	51.34%	36.79%	13.53%	1.83%	0.03%
	24	95.92%	92.00%	84.65%	71.65%	60.65%	36.79%	13.53%	1.83%
	48	97.94%	95.92%	92.00%	84.65%	77.88%	60.65%	36.79%	13.53%
	96	98.96%	97.94%	95.92%	92.00%	88.25%	77.88%	60.65%	36.79%

spective of the surviving subscribers (users). Consequently, the performance received by the surviving user community needs to be analyzed. The analysis must take into consideration any increase or decrease in infrastructure activity resulting from the organizational response to the event being studied. As an example, the performance of file transfers across an enterprise may average 100 megabits per second (Mbps) under normal circumstances but a catastrophic failure may reduce the performance to 10 Mbps even if there is no loss of service (i.e., availability).

- **Reconstitution Time.** Once a catastrophic event has taken place, the time required to resume mission-critical activities is one of the most important residual measures for describing an IT infrastructure's survivability. Calculations of reconstitution time should take into consideration both emergency recovery plans and impairment of recovery capabilities caused by the event.

Comparing Architectures

In evaluating alternative architectures, some composite measures across a set of potential events are often useful. In this section, we suggest two methods to compare the survivability of alternative architectures.

Develop a Conical Event Set

A conical set of events is a set that includes all the important types of catastrophic events that the IT infrastructure should be designed to survive. For example, if fires, floods, viruses, and power outages are the types of events that must be anticipated in the IT infrastructure design, then the conical set of events includes at least one example of each type. Ideally, the conical set of events includes the worst case example of each event type.

Compare the Survivability of Architectures

After calculating the residual metrics for each of the events in the conical set and alternative architectures, it may be desirable to make an objective comparison of the survivabilities. We discuss two conceptual ways of making such comparisons.

- **Multiple Criteria Methods.** Within the specialty area of multi-criteria optimization, the concept of best alternatives has been formalized. This concept simply states that if one alternative has scores that are greater or equal to the corresponding scores of all other alternatives and has a unique best score for at least one criterion, it is clearly the best alternative. Consequently, if one of the architectures being evaluated has higher residual availability for all events, lower reconstitution time for all events, and fewer single points of failures for all events, it is clearly the best architecture among those being compared.
- **Weighting Methods.** In situations where there are many criteria that can be weighted by some subjective means and no clear best alternative based upon multiple criteria methods exists, a reasonable approach to selecting the most survivable alternative is to create a survivability index. The index would be a simple, weighted sum of the criteria c_{ij} values for each catastrophic event multiplied by their respective weights, w_{ij}. An index value is computed for each alternative and the alternative with the highest (or lowest) index is selected as the best alternative. The formula is:

$$\text{Survivability Index} = \sum_i \sum_j w_{i,j}\, c_{i,j} \qquad (9)$$

In (9), the *i* index ranges over the set of catastrophic events, and the *j* index ranges over the survivability criteria.

Summary

Availability and reliability are well-established disciplines upon which SLAs are frequently established. However, survivability is an increasingly important factor in the design of complex systems. More effort is needed for survivability to achieve the same rigor as enjoyed by availability and reliability.◆

References

1. Patterson, D., et al. "Recovery Oriented Computing: Motivation, Definition, Techniques, and Case Studies." Berkeley, CA: University of California Berkeley, Mar. 2002.
2. Cisco Systems. "Service Level Management Best Practices White Paper." San Jose, CA: Cisco Systems, July 2003.
3. Ellison, R.J., et al. "Survivable Network Systems, an Emerging Discipline." Technical Report CMU/SEI-97-TR-013, 1997.

About the Authors

Jack Murphy, Ph.D., is president and chief executive officer of DeXIsive Inc., an information technology system integrator focusing on information assurance, network solutions, and enterprise infrastructure applications. Prior to this, Murphy was the chief technical officer for EDS U.S. Government Solutions Group. He is an EDS Fellow Emeritus and remains actively involved with EDS thought leaders. Murphy retired as a lieutenant colonel from the Air Force where he spent much of his career on the faculty of the Air Force Academy teaching math, operations research, and computer science. He has a doctorate in computer science from the University of Maryland.

DeXIsive, Inc.
4031 University DR STE 200
Fairfax, VA 22030
Phone: (703) 934-2030
Cell: (703) 867-1246
E-mail: jack.murphy@dexisive.com

Thomas Ward Morgan, Ph.D., is a chief simulation scientist at CACI International, and leads the World Wide Engineering Support Group Modeling and Simulation team. He has been involved in networking and software development since 1970. Morgan served in the Army Medical Service Corps and worked on the Automated Military Outpatient System and Next Generation Military Hospital projects. He has also worked at AT&T Bell Laboratories participating in the design of internal corporate networks to support AT&T's customer premises equipment operations. He was active in the Institute for Operations Research and the Management Sciences and has published over 60 technical papers. He has a Master of Science and a doctorate in electrical engineering.

CACI Federal
14111 Park Meadow DR STE 200
Chantilly, VA 20151
Phone: (703) 802-8528
E-mail: wmorgan@caci.com

Why Isn't There an "*I*" in *Team*?

I'd like to start off this column with a story about a friend of mine. We'll call my friend "Jeff." The story involves his daughter, who we will call "Jill." It seems that Jill, who is an accomplished swimmer, once had a contest with a friend of hers to see who could hold her breath the longest. Jill easily won the contest – a fact that had to be verified by her friend – as Jill had the endurance and willpower to hold her breath so long that she passed out. I also hear that ambulances were involved afterwards.

You know, as a father, I would be proud to have a daughter like Jill[1]. I know Jeff is! I imagine anybody with that kind of willpower is rather immune to peer pressure, and will also ultimately succeed in whatever she sets her mind to. She definitely is tenacious – which is a synonym for stubborn. Obviously, she has the makings of a fine engineer.

I have long thought that those children who have the potential to be truly gifted engineers could be identified easily in playschool. All it requires is a sandbox and a few toys. Insert children. There will be a few children who will happily play with others, enjoying the sand and the toys. These are not future engineers. Over in the corner of the sandbox, however, there will be a child happily playing alone, by himself or herself. Whenever any of the other children come too close, this child will throw sand at the intruders, and may go as far as knocking intruders on the heads with the sand-building instruments, all the while clutching at his/her toys shouting, "Mine!" This is a potential engineer!

Let's face it – for developers, a good chunk of our day is spent sitting alone in cubicles, ignoring the hustle and bustle surrounding us. We occasionally put on earphones and listen to music, or just learn to tune out surrounding noise. And there we sit – oblivious to the outside world, creating "stuff." The problem, as any good software tester will tell you, is that putting your stuff together with my stuff will uncover interface errors. And again, as any good tester will tell you, interface errors are numerous and often difficult to fix. From my point of view, the problem is your code. You, of course, might have a slightly differing opinion. Many testing authorities say that up to 75 percent of all errors stem from interface problems.

It's not all our fault; we're trained to operate in isolation, starting in college. We are encouraged to come up with individual, innovative solutions to problems. The problem is that your individual, innovative solution might not make a lot of sense to me, and I have to write the software that interacts with yours. To make my software interact with your software, we need to communicate before and during the software creation. This can be done via the tried-and-untrue method of meeting after meeting after meeting – but this method hardly ever works. What I need is a way to encourage good teamwork and cooperation, without spending 50 percent to 70 percent of my day in meetings. Enter Personal Software Process℠ (PSP℠) and Team Software Process℠ (TSP℠).

Granted, I am a PSP/TSP zealot. PSP encourages good individual programming practices, and TSP creates an environment that permits motivated individuals to work as an efficient team. I have been teaching PSP/TSP since 1998 – and it's one of the few tools that I teach to developers that (almost) always prove beneficial. Why almost? Because some developers just don't want to interact as a team. They don't have the interpersonal skills to play well in the sandbox. Part of this is because of insecurity ("Everybody else is so much better than me – and I don't want anybody knowing this."). Part is because they never learned good team-building skills. (Note that spending a full day at an off-site team-building exercise where you learn to fall backwards into your teammates arms is not really that useful). However, I have found that the majority of developers I have worked with do have the skills and motivation to become part of an integrated team.

PSP/TSP works because teams are inherently more effective than individuals. The so-called synergistic effect really works – where the combined action of two individuals is more than the sum of their individual efforts. Developing software through teamwork is nothing new, but what makes PSP/TSP effective is that it includes quantifiable metrics that allow developers, as a group, to accurately plan and track their progress. These team metrics make the team effective, because, as Lord Kelvin said back on May 3, 1883, in a lecture to the Institution of Civil Engineers:

> When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of a science. [1]

My personal translation of this – which I have repeated to every PSP class I have taught – is, "If you can't count it, you can't account for it." But Lord Kelvin only mentions the need for metrics, not teamwork. If only I could find a reputable source to support teamwork:

> Two are better than one, because they have a good return for their work: If one falls down, his friend can help him up. But pity the man who falls and has no one to help him up! ... Though one may be overpowered, two can defend themselves. [2]

— David A. Cook, Ph.D.
The AEgis Technologies Group, Inc.
<dcook@aegistg.com>

References

1. Kelvin, Lord <http://www.cromwell-intl.com/3d/index.hml>.
2. The Bible. Ecclesiastes 4: 9-10, 12 (NIV)[2].

Notes

1. In case any of my daughters read this column – yes, you are "tenacious," also. And I am proud of you.
2. The biblical scholars among you might note that I skipped verse 11: "Also, if two lie down together, they will keep warm. But how can one keep warm alone?" *This* is not the kind of teamwork that I either encourage or condone among my coworkers!

CrossTalk / 309 SMXG/MXDB
6022 Fir AVE
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737